2003

# Constructing a reproducible testing environment for distributed Java applications.

Kun Wang
*University of Windsor*

# Constructing a Reproducible Testing Environment for Distributed Java Applications

by

**Kun Wang**

A Thesis Submitted
to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for the
Degree of Master of Science at the University of Windsor

Windsor, Ontario, Canada
October 2003

*Your file Votre référence*
*ISBN: 0-612-86705-6*
*Our file Notre référence*
*ISBN: 0-612-86705-6*

# Abstract

The emergence of the global Internet, wireless data communications, and the availability of powerful computers is enabling a new generation of distributed and concurrent systems. However, the inherent complexity of such systems introduces many new challenges in system testing and maintenance. One of the major problems in testing such systems is that executions with internal non-deterministic choices make the testing procedure non-repeatable. A natural solution is to artificially force the execution of a program to take desired paths so that a test can be *reproduced*. However, with geographically distributed processes and heterogeneous platform architectures, distributed systems have imposed new challenges in developing effective techniques for *reproducible testing*.

The goal of this research is to build an environment to automate testing for distributed and concurrent Java applications. We will focus on controlling the order of occurrences of input and remote call events according to a user-specified test scenario, which is composed of input data, a constraint expressed as a partial order over the input and remote call events, and expected output. The testing environment is by itself distributed and does not require source code intrusion into the application under test. With minor changes, the testing components can also be reused in CORBA-based applications implemented in Java.

**Keywords:** Distributed Systems, Nondeterminism, RMI, Specification-Based Testing, Reproducible Testing, CORBA, Portable Interceptor, Middleware, Concurrent Program, Dynamic Proxy, Reflection.

## Acknowledgement

This work would not have been possible without the help of many people.

First and formost, I would like to express my sincere gratitude to my research supervisor, Dr. Jessica Chen for giving me the opportunity to conduct research in this area and for her constant guidance throughout its duration. I am very thankful for her vision, advice, patience and serious attitude that has been an invaluable source of support throughout my graduate program.

I would also like to thank my committee members, Dr. Christie Ezeife, Dr. Xiang Chen and Dr. Ngom Alioune for spending their precious time in reading my thesis and putting on their comments and constructive advice on this work.

My special thanks go to the graduate secretary of the School of Computer Science, Ms. Mary Mardegan for her consistent help. Thanks also go to my peers, Songtao Chen and Yongdong Tan for their valuable advice and interesting discussions.

# Table of Contents

## List of Tables

## List of Figures

# 1. Introduction

## 1.1 Motivation

With advances in networking and middleware technologies and web support, distributed systems are gaining increasing popularity in their use, ranging from various industrial communication systems to our daily social assistance and control systems such as education systems, healthcare systems and transportation systems. However, the inherent complexity of distributed and concurrent systems has imposed various difficulties to both software development and software maintenance. Heterogeneity in terms of the adopted hardware, platforms and implementation languages, and nondeterminism existing during the executions are typical sources of such difficulties. Heterogeneity in distributed systems poses many difficulties in system communications and interactions. Due to nondeterminism, the behavior of a distributed program is no more predictable: running a program several times with the same input may not guarantee the same result. This is because a distributed and concurrent system usually has many different execution paths due to the fact that different processes are running at different speeds, with various kinds of process cooperation, which leads to different interleavings of the execution paths because of the interactions among different processes. As a consequence, testing turns out to be non-repeatable.

When testing a sequential program, if we observe a certain erroneous phenomenon during a testing procedure, we usually execute the program again with the same test input to repreat the erroneous execution or to collect debugging information. This is called *test replay*. After we have modified the program, we can run it again with new test input as well as with previously tested input to verify that the detected errors are removed and that no new errors are introduced. This last testing step, called regression testing, is especially needed for software maintenance. When testing concurrent programs, since a test may not be repeatable (meaning that it is not guaranteed that we can obtain the same output when running such a program several times with the same input), we may not be able to see the error again or to locate the buggy code, if we observe an error during or after a program execution. In particular, during regression testing, we may not be able to check whether the errors are corrected, neither can we ensure that no new errors are introduced after this program is updated.

## 1.2 Overview of Possible Solutions to the Nondeterminism Problem

A natural way to tackle the above problem is to *direct* the program execution so that with a given input, we can artificially enforce some of the internal execution choices [2, 6, 7] on a concurrent program. If well-controlled, the execution of a concurrent program can be *directed* and thus, the observations can be *reproduced*. In fact, people have developed various techniques to control the program executions for both debugging and testing purposes. For reproducible testing, we assume that we are given a set of test scenarios, which consist of not only test cases but also some path constraints. The test case, as usual, describes the external input, i.e. a sequence of input data to each process, and the expected observations (outputs). The additional path constraints describe some further constraints on the execution paths with the given test input. Such path constraints can be expressed as a partial or total order among external input events in the test cases and some internal events such as certain statements in the program. Obviously, the path constraints are often designed to denote the typical or representative scenarios in which possible errors or bugs may reside.

Unlike in a debugging approach where we define the checkpoints individually, in an automated reproducible testing, we can predefine in general the events we are interested in controlling their order of occurrences. Typically, we consider three types of events:

✓ The synchronization events [1-3, 6, 7, 19]

This is based on the observation that different output of multiple executions of a distributed and concurrent program with same inputs are often caused by the different orders of accessing shared objects (synchronization events) by various processes.

✓ The input events

In a distributed system, the orders of the input events may also be a source of different observable behaviors. Of course, we cannot define orders among input events in the same process, since the order of input events within an individual process is deterministic. The constraints we will add here are over the orders of input events among different processes.

✓ The inter-process communication events [2, 3, 6, 7]

An inter-process communication event can be viewed as an external input to the target process of the event (also called remote events). As a result, the orders of such events also contribute to the different behaviors of the overall distributed system.

There are two main issues in automating software testing. One is to automate the generation or partially automate the generation of the test cases and path constraints; and the other is to realize the control of executions of an Application Under Test (AUT), i.e., force the program execution according to the specified paths with given input. In this work, we only consider the latter issue, i.e.: we will only consider implementing automated control over input events and inter-process communication events in realizing a reproducible testing, and assume that both test case and path constraints are given.

Automated testing naturally requires some software instrumentation techniques, which monitor, analyze and manage the executions of processes or the interactions among different processes and their running environment in a software system. The instrumentation can be realized via two approaches: 1) via intrusion into the source code of an AUT, and 2) via the interception service in the underlying runtime system. Much of previous work on software instrumentation focuses on the source code intrusion technique and can be broadly divided into two groups. One is to integrate source code and test code. This is the so-called built-in test method [9, 14], by which we have the program source code and testing code in an integrated form to enhance software maintainability and traceability. The other approach is to extend source code with additional process communications. Along this line of research, the source code of the AUT is augmented by some communication constructs between the AUT and the automated test control system [1-3, 6, 7, and 19]. This is of particular interest when we intend to gain some control over the internal non-deterministic choices in the AUT.

Although traditional approaches provide effective techniques to software instrumentation for automated testing, they also have some major disadvantages. For example, in the built-in test technique, tests are constantly occupying space while most tests will only be used once when the component is deployed. In addition, with much test code integrated with the program source code, the readability of such a program is seriously affected. Extending source code with additional process communications assumes the availability of source code of an AUT, which is usually impossible for today's commercial off-the-shelf (COTS) software components. In addition, the behavior of the extended code may deviate from the original AUT, which raises consistency problem. In this research, we will solve the above-mentioned problems (e.g.: poor readability and unavailability of source code) by building software instrumentation into the application's run-time environment so that an application itself does not have to contain any testing code at development and deployment stages and the application source code can remain untouched during system testing or maintenance.

Recently, middleware technologies have been widely adopted to develop large-scale and complex distributed applications. Middleware technologies such as CORBA, Java RMI and DCOM provide us with core software infrastructures that make it relatively easy to build distributed applications that are of high-performance and are scalable. They also offer a set of services that support component interoperability in a heterogeneous environment, while hiding the details of its network management and communications. The advances of middleware technologies provide us with new opportunities to explore the second approach to software instrumentation for distributed applications, especially for process communications across machine boundaries -- integrate the instrumentation into middleware layer. Software instrumentation into middleware level is a novel approach that we will adopt in our research. It is superior to the code intrusion technique in that it requires neither the availability of the source code nor test user's knowledge about the AUT. It is built independent of the implementation of an AUT, thus the AUT can remain completely as a black box. Now a question arises up: how can we implement this instrumentation into the middleware, in order to monitor, and further control the executions and interactions of processes in an AUT?

In CORBA, this can be accomplished by way of CORBA *portable interceptors*, which are actually some hooks into the CORBA ORB. They allow users to insert their own code into the ORB and intercept the normal flow of program execution without changing either the applications or the ORB implementation. This user-provided code is invoked at certain interception points during remote request/reply processing, and thus can be used for inspecting and manipulating the remote requests and responses.

## 1.3 Objective and Contributions

In this thesis work, we consider using distributed Java applications communicating via Remote Method Invocation (RMI) as the AUT. We have chosen distributed Java applications based on a number of reasons. First, Java is becoming increasingly popular in developing network based, distributed and concurrent software systems because of its portable, easy-to-use, and security features. Second, most distributed and concurrent applications involve a set of processes executing in parallel, with each process having multiple threads running concurrently. This characteristic of distributed and concurrent programs leads to two requirements in this work: 1) developing a typical AUT, which can closely model the behavior of a distributed and concurrent program (which has distributed processes and multiple threads within each process), and 2) constructing a testing control environment that is able to handle multithreading issues. We use Java because Java language provides a built-in facility to support multithreading. This support is a nice feature in that you do not have to think about the low level mechanism for partitioning system resources (such as CPU time) for multiple threads, since this is done by Java, which makes programming with multiple threads a much easier task. We have chosen RMI as the underlying communication mechanism since RMI is a distributed object model that allows programmers to develop distributed Java programs with the same syntax and semantics as those that are used for non-distributed programs. It offers a middleware (similar to CORBA ORB) by which distributed processes can communicate with each other and pass information back and forth.

Unlike CORBA *portable interceptors*, however, the major limitation in Java for software instrumentation is that it does not contain such an interception mechanism. So, in this thesis work, we first provide a solution to injecting an interception service into the underlying RMI middleware. This service is similar to the *portable interceptors* in CORBA, which is to "peek" the executions and communications among processes of a distributed Java program and intercepts Java remote method requests and responses for the control of remote calls. This interception service is achieved by making use of Java Reflection to modify and extend existing Java libraries.

The testing environment constructed in this research is by itself distributed, with some of its testing components residing within the same host as each process in the AUT (thus called local testing components). These local testing components include a single path controller and a local test driver for each process on each machine. To distribute the test controllers and test drivers to be local is for efficiency reason. Whenever a thread in a process needs to interact with threads in other processes (i.e., a remote event), this event will be intercepted by the middleware, which will send a request to its local controller on behalf of this thread. This controller is responsible for deciding whether this thread should proceed, wait for other threads, or resume from a waiting state. In general, a *test driver* is a program that performs test setup, makes a sequence of calls to the software component under test using a different range of test data for each call. The driver will normally record output data to a file for use in examining the results of each test run [29], and then do necessary clean-up tasks. A test driver in the context of this work is specific to a single process and is responsible for starting a process under test, feeding inputs to it, recording its result and then sending this result to the centralized test oracle (a program for checking test results against expected results. See Chapter 4 for detailed definition) for verification. Whether a driver could proceed to feed an input to its process also depends on the permission from its test controller, which makes this decision according to the path constraints and the current overall status of the AUT. Deploying these components in a single host will definitely reduce network delay caused by a lot of communications among these components. A centralized communicator is also used to coordinate among test controllers. This communicator is simply a "broadcaster", which accepts updates of each process's running states from each path controller and broadcasts them to all other controllers. For the communications between testing components, we have also adopted Java RMI.

By means of the interception services, we can hook up the above testing environment into the Java RMI implementation, and further control the order of occurrences over the local input events and remote method invocation events of the AUT.

## 1.4 Thesis Structure

This thesis work is organized as follows: Chapter 2 reviews some previous work in automated reproducible testing, and discusses their disadvantages. In Chapter 3, we propose our approach to automated reproducible testing for distributed Java applications. We also define the format of a test scenario, including the formats for events, test constraint, and test oracle, and show a motivating example – Online Conference, which will be used as the AUT in later chapters. In Chapter 4, we briefly review the concept and architecture of Java RMI, and discuss in detail how the interception service is injected into the RMI implementation in order to provide a mechanism to hook up the testing control transparently to user applications. Chapter 5 shows the architecture of our proposed testing environment and discusses the functions of each testing component. We describe how this testing environment works with the Online Conference as the application under test. This chapter also introduces the control algorithm and the automation of the testing environment setup. In, Chapter 6, we overview the CORBA application architecture, CORBA middleware - ORB, OMG IDL and Java IDL, which are some prerequisites to develop distributed applications based on CORBA architectures. This chapter also introduces the CORBA *Portable Interceptors,* an essential technique to realize interception service in CORBA applications. Chapter 7 describes in detail how to reuse the testing components for distributed Java applications in a CORBA environment, and compare the similarities and differences between these two testing environments in a variety of aspects. In Chapter 8, we run several experiments to evaluate the functionality and performance of our testing environment. Chapter 9 concludes this thesis work and indicates possible future work in related areas.

# 2. Related Work

Previous work on automated reproducible testing usually involves three major issues: 1) how to define the format of a synchronization event, which should contain sufficient information to determine an event and direct the execution of a program according to a given test scenario, 2) how to collect the test constraints and test cases. The set of test scenarios must be small enough to be exercised in a relatively short period of time, yet be adequate enough to uncover all potential program errors, and 3) how to develop a tool to repeat the previous execution of a concurrent (and possibly distributed) program based on a given test scenario (reproducible testing). This issue usually requires the introduction of some control mechanism with the help of software instrumentation techniques. Our future research will focus on the first and third issues. We assume that the test constraints and test data are available during a testing process.

During the past years, a lot of research has been done on above three issues. Especially, many quite efficient techniques have been developed to automatically generate test data and test constraints [7, 20-25]. However, as mentioned before, much of previous work on software execution control and instrumentation still relies on the source code intrusion technique. This work can be broadly divided into two groups: integrating source code and test code (built-in-test method), and extending source code with additional process communications.

## 2.1 Built-in-test Techniques

In the first approach, the testing code is integrated into the program at design and implementation stages as member functions, class clusters or sub-systems to improve software testability. Such an augmented program can run either in normal mode as a conventional program or in testing mode for testing and maintenance purposes. This method draws attention to build testability into objects and frameworks, so that the software testing and maintenance can be self-contained. The most interesting feature of the built-in-test techniques is that tests can be inherited and reused in the same way as that of code in conventional object-oriented frameworks [9]. A prototype of a built-in-test object in C++ code is given below:

```
Class class-name {
            // Interface
            Data declaration;
            Constructor declaration;
            Destructor declaration;
            Function declarations;
            Tests declaration;      // Built-in test declarations

            // Implementation
            Constructor;
            Destructor;
            Functions;
            TestCases;              // Built-in test cases as
                                    // new member functions (methods)
} TestableObject;
```

Figure 2.1: An Object with built-in tests [9]

In this prototype, the test declarations in the interface and the test cases in the implementation have been embedded into a standard object structure. In this way, the built-in tests may be inherited and reused in the same way as that of standard and application specific member functions within the object. The built-in-test object component has the same behavior as that of the conventional objects when normal functions are called. But if the built-in tests are called as member functions in the object, e.g.:

*TestableObject :: TestCase1;*

...

*TestableObject :: TestCaseN;*

the built-in-test object can be automatically tested and corresponding results are reported. The same built-in-test method can be extended to the class cluster or object-oriented framework levels. Built-in tests in the class cluster level are a set of class files acting as test files in an OO sub-system, while built-in tests in the object-oriented framework level are a set of tests playing as a sub-system in the whole OO framework. In this way, tests can be built in all components of software in the scopes of objects and systems. The maintainability of software can be increased by the possession of the features of self-containment of code and tests. Further details of built-in tests in class cluster and framework levels can be found in [9, 14].

## 2.2 Control of Nondeterminism in Concurrent and Distributed Systems

Previous research in the second approach in support of control of non-deterministic behaviors of concurrent and distributed systems has also been extensively conducted. This approach centers on augmenting the source code of an AUT with some communication constructs between the AUT and the automated test control system. This is of particular interest when we intend to gain some control over the internal non-deterministic choices in the AUT and force the system to take particular execution paths [1-3, 6, 7, and 19]. The idea of deterministic testing of concurrent programs was first introduced by Kuo-Chung Tai and Richard H. Carver in [1, 19], where the non-deterministic behaviors of concurrent programs are considered as the results of the unpredictable progress of concurrent processes accessing synchronization constructs (thus so called synchronization events). They presented a language-based approach, where programming language supported synchronization constructs such as semaphores and monitors are used to deterministically test and debug concurrent Ada programs. Again, based on the assumption that the test data and constraints are given during testing, we can summarize this approach as three steps: 1) defining the format for synchronization sequences, which provide sufficient information for test control and deterministic execution, 2) transforming a concurrent program into a slightly different program in the same language, which is equivalent to the original one except that some statements are inserted (using a tool like a parser) right before and after synchronization events, and 3) developing a synchronization sequence replay tool to control the execution of the transformed program so that an execution of this program deterministically exercises a given synchronization sequence. Although they implemented the reproducible testing in

Ada, the idea works well for other languages supporting synchronization constructs, and the transforming tool and replay tool are relatively easy to develop.

### 2.2.1 Control of Communication Events

Along this line of research, D. Kung et al. [2, 7, and 28] proposed a state-based reproducible testing technique in a distributed environment by adopting CORBA. They not only defined a replay control mechanism (mentioned in the Introduction), but also described an algorithm to automatically generate test sequences. This test sequence generation technique is realized by constructing atomic state machines (ASMs) for interesting single shared variables, and composite state machines (CSMs) when more than one shared variables are used to describe the state behaviors of a program, and then building a test tree based on these ASMs and CSMs to generate all possible test sequences. In their approach, the generated test sequences by their state-based algorithm are total orders of remote call events, whose number is very large even for a relatively small program.

However, we do not have to specify totally ordered event sequences as path constraints, because the orders of some events that are in an individual process are pre-defined and can be identified in a formal design specification. So in our research, we will only consider the partial orders among input and remote call events as our test constraints.

### 2.2.2 Control of Synchronization Events

Based on the fact that non-deterministic behaviors in a concurrent program usually arise from concurrency-related statements, many researchers have proposed approaches to controlling the orders of synchronization events when testing a concurrent program. In [3], X. Cai and J. Chen presented the framework of an automated test control toolkit, which can artificially control the partial order of synchronization events in a distributed multithreaded programs. This framework adopts CORBA infrastructure as its underlying middleware for communications among processes, and the implementing language of this framework is Java. In Java language, each object with synchronized method or synchronized block is associated with a monitor, and an operation (method invocation) on a monitor is defined as a synchronization event. By introducing constraints on the orders of such events, and extending the source code of a program with additional communication constructs, this work realized control of some important synchronization events.

Although traditional approaches provide some effective techniques to software instrumentation for automated testing, they do have some major disadvantages. For example, in the built-in test technique, tests constantly occupy space while most tests will only be used once when the component is deployed. Moreover, because tests are built into an application at design and implementation phases, this kind of test also burdens application developers with the test design and implementation issues. The methods proposed in [1-3, 6, 7, and 19] isolate testing from the software component development stages, and will leave component developers free of the concerns about the testing during design and implementation phases. However, they all assume the availability of source code of an AUT, which is usually impossible for today's commercial off-the-shelf software components. When software components are issued to the markets, they are often in binary forms, and application developers (and testers) who will use (and test) these components do not have access to the source code of those components due to copyright issues. Even if they have the source code and can transform them with some tools, the behaviors of the extended code may deviate from the original AUT, which raises consistency problem. Furthermore, with additional language constructs integrated with the program source code, the readability of such a program could be seriously affected.

# 3. Proposed Approach

## 3.1 Advantages of the Proposed Approach

With the above-mentioned testing problems and disadvantages of previous work, the objective of this research is to present an environment that can realize the automated control in reproducible testing where the AUTs are distributed Java applications communicating via RMI, and the path constraints are defined as partial orders over the input events and remote method call events. Unlike the software instrumentation techniques proposed in the previous research [1-3, 6, 7, 19], the test control is based on constraints on local i/o events and remote method invocation events, and the instrumentation in our testing environment does not require any source code intrusion and is completely transparent to both client and server programs.

To force an AUT to satisfy a certain path constraint, we need to introduce some control mechanism into the system during the execution. The execution of the AUT is augmented by additional communications between the control mechanism and all the processes in the AUT. In this thesis work, we will build this communication mechanism into the underlying Java Run-time environment. How to alter the execution of the underlying environment depends on the type of events we are interested in controlling their timing of occurrences. For input events, we employ the test driver, which is in charge of starting the AUT and providing input to a process, and to actually carry on its task on a real-time basis (see Section 5.2 for details of how the test driver control the order of input events). For the remote method call events, on the other hand, we can specifically inject this communication mechanism into the middleware layer of the RMI implementation, and further hook up a control meachanism by means of this injected communication.

Compared with previous software instrumentation techniques, our approach provides the following benefits:

- Intrusion into the underlying system requires neither the availability of the source code nor test user's knowledge about the AUT. Thus, the AUT can remain completely as a black box. By providing an interception service, application testers are allowed to plug in their testing code into the RMI middleware layer in a "standard" and "systematic" way, in the sense that users can inject their own code into the RMI middleware by implementing and extending a set of CORBA-like classes, and starting this interception service in a way very similar to that of CORBA.

- Our test control environment is distributed and scalable. We do not limit the size of an AUT in terms of the number of its processes. In fact, the AUT could consist of any number of processes, which may run in different hosts and operating systems during each test. By distributed testing environment, we mean that a test controller, a test driver together with an interceptor server (see definition in page 28) are deployed locally with each process on a single machine in a multi-process (and thus multiple machines) application, and a central test oracle and communicator can be installed on other machines.
- A program can be run either in normal mode or in testing mode. As per the first advantage, the software instrumentation and testing components are independent of specific application and its implementation. So, the underlying instrumentation will not affect the normal execution of an AUT if it is not turned on. Users are also given the flexibility to choose to turn on either the client or the server side (or both) instrumentation, or to dynamically turn the interception service on or off during a testing procedure, by implementing and extending some standard classes, then starting the supporting services or executing some pieces of code to register/un-register their interceptors.
- The testing components, i.e., test controllers, drivers, RMI interceptor servers, test oracle and communicator can be reused in CORBA architecture with only minor changes.

Other than software testing, intrusion into the underlying run-time system can also be used in software instrumentation technique in support of software debugging, monitoring and resource management etc. For example, Friedman and Hadad in [33] have discussed the instrumentation in existing CORBA ORB implementation for caching, load balancing, and redundancy in assuring the reliability of real-time software systems. In [8], Denis Reilly and A. Taleb-Bendiab use the Java Dynamic Proxy technique to build some kind of interception services into the underlying Jini implementation, and further proposed a service-oriented, dynamic instrumentation framework that provides support to monitor and manage Jini applications. Similar to this work, our instrumentation provides a monitoring framework for dynamic analysis of distributed Java applications, enabling tracing of flows of control transparent to application developers, and further managing individual components, their running environment and their interactions.

## 3.2 Test Scenario and a motivating example

As mentioned in the introduction, we consider the automated control over the timing of two types of events: the input events and the remote call events. This is reflected in the definition of our test scenarios. Let $V$ be a set of input/output values, $I$ be a set of remote interface names and $M$ a set of remote method names. A test scenario is defined as an element of $TS = \langle E1 \cup E2, C, O\rangle$ where

- $E1 = \langle N, V, N, \{“i”, “o”\}\rangle$ ($N$ is the set of natural numbers) is the set of i/o events. $(j, v, k, s) \in E1$ denotes input value $v$ to process $j$ for the $k$th time (when $s$ = “$i$”) or receive an output value $v$ from process $j$ for the $k$th time (when $s$ = “$o$”).
- $E2 = \langle N, I, M, N, \{“qc”, “qs”, “ps”, “pc”\}\rangle$ (N is the set of natural numbers) is the set of remote call events. $(j, i, m, k, s) \in E2$ denotes an event of calling method m on the interface $i$ from process $j$ for the $k$th times, at the time of $s$ where:

  – $s$ = “$qc$”: when the call request is at the caller’s side;
  – $s$ = “$qs$”: when the call request arrived at the callee’s side;
  – $s$ = “$ps$”: when the call response is at the callee’s side;
  – $s$ = “$pc$”: when the call response arrived at the caller’s side.

- $C \subseteq (E1 \cup E2) \times (E1 \cup E2)$ is a binary, transitive relation between events to denote the ordering constraint among them. $(e1, e2) \in C$ means that we require $e1$ to *happen before* $e2$.

- $O$ is a boolean expression that we expect to be true (test oracle). It may contain pairs from $E1 \times E1$ that shows the happen-before relationship between two i/o events.

Let us consider an application of on-line conference control. With the use of Internet and multimedia, it is possible to host an on-line conference. Now let us consider using the distributed bakery algorithm introduced in [4] to guarantee that only one person can speak (enter his critical section) at a time. Distributed systems involving multiple processes usually compete to use shared data. A *critical section* is a code segment in each process, in which shared data may be accessed. Each process executing its critical section must gain exclusive access of the shared data and ensure that only one process is allowed in its critical section at any time.

The distributed bakery algorithm goes in this way: *n* processes (representing n people) communicate with each other in a peer-to-peer manner in order to enter a critical section (to speak). Whoever wishes to enter the critical section should pick up a ticket number, broadcasts this number together with its process id to all other processes, and wait until it has received responses from each other process that the chosen number becomes the lowest. To realize this, each process maintains a local number (e.g., High_Number) that is what *it knows* so far the biggest one among all the numbers maintained by various processes. Initially, this High_Number is set to the same value (e.g., 0) in each process. When receiving an input signal of *willing to speak*, the process locally picks up a number that is 1 greater than the High_Number (i.e. High_Number + 1) and sends a request with this number to all other processes. Each process which receives a request together with a ticket number smaller than its own local chosen number will reply immediately, meaning it allows the sender to enter the critical section. On the contrary, if a process who receives a request together with a ticket number (Received_Number) is greater than its own local chosen number, it will suspend its reply until it has exited its critical section. In either case, the High_Number of this process will be reset to value of the the original High_Number or the value of the Received_Number, whichever is greater. The request sender will enter the critical section only after it has received replies from *all* other processes.

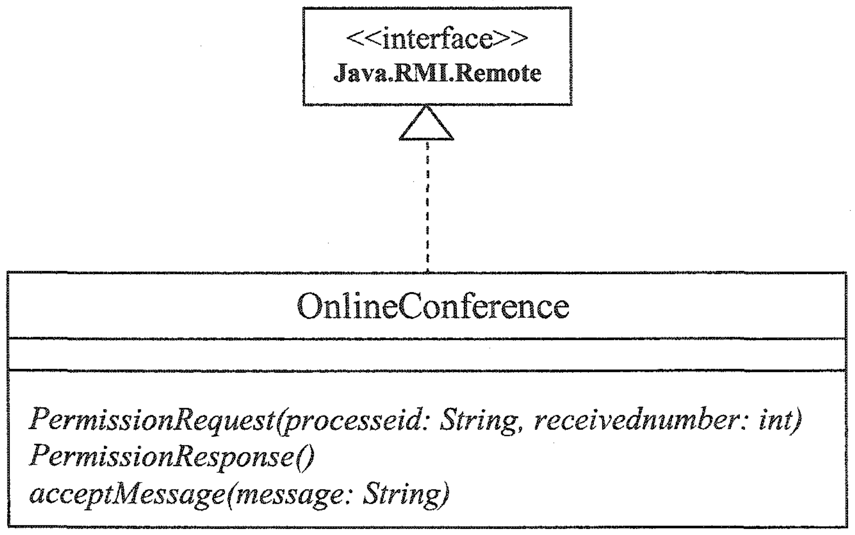

<<interface>>
Java.RMI.Remote
OnlineConference
PermissionRequest(processeid: String, receivednumber: int)
PermissionResponse()
acceptMessage(message: String)

Figure 3.1 The RMI remote interface definition for the on-line conference example

Figure 3.1 shows the definition of the remote interface *OnlineConference* in terms of UML class diagram. When receiving an input indicating *a willing to speak* from the test driver, a process locally picks up a number and makes a series of remote calls (*permissionRequest*) of all other processes with its own process identifier and its own number, in order to get permission from those processes to enter its critical section. Correspondingly, whenever allowed, a process remotely calls *permissionResponse* of the requesting process to grant it such permission. After obtaining permission from all other processes, the requesting process remotely invokes the *acceptMessage* of other processes to broadcast its messages. One of the typical scenarios that we are interested in testing here is when two participants wish to speak at the same time. More precisely, we want to test whether the program works correctly when two individual processes locally pick up the same number. Apparently, *two individual processes locally picking up the same number* is an important case when potential concurrency-related design or implementation errors may show up. However, this is impossible with a traditional testing technique where we consider only the input to the program and the corresponding output from its execution, because here we need to gain the control over the execution of the two processes. With the present testing technique, the desired scenario can be realized by controlling the timing of occurrences of the user's input (*willing to speak*) and some remote method calls during the execution. Let us consider a test scenario where there are two processes (representing two users): each user requires speaking only once, and both users pick up the same number. We can define the test scenario in the following way:

- *V* = *{*0, 1*}*: There are two kinds of input events in this example. One is to signal the request to speak (input value 0), and the other is the signal of the end of speaking (input value 1). There are also two kinds of output events. One is to denote the starting point of speaking (input value 0), and the other is the actual end of speaking (input value 1).

- *I* = *{"OnlineConference"}*: In this example, there is only one remote interface *OnlineConference,* for which a remote object will provide implementation.

- *M* = *{"permissionRequest", "permissionResponse"}*: There are two remote methods defined in the remote interface *OnlineConference.*

Recall that the set of i/o events and the set of remote method invocation events can be respectively represented as *(ProcessId, Value, Number, S)* (where *S* can be either *"i"* or *"o"*) and *(ProcessId, InterfaceName, MethodName, Number, S)* (where S can be one of the four constants: *"qc"*, *"qs"*, *"ps"*, and *"pc"*), the test scenario can be further elaborated as follows:

- *ie*1 = (1, 0, 1, "*i*"): event *ie*1 is the first input of value 0 to process 1.
- *ie*2 = (1, 1, 1, "*i*"): event *ie*2 is the first input of value 1 to process 1.
- *ie*3 = (2, 0, 1, "*i*"): event *ie*3 is the first input of value 0 to process 2.
- *ie*4 = (2, 1, 1, "*i*"): event *ie*4 is the first input of value 1 to process 2.
- re1 = (1, "OnlineConference", "permissionRequest", 1, "qc")*: event* re1 *is the first remote call of method* permissionRequeset on the interface OnlineConference *from* process 1 *when the call request is still on the caller's side (i.e.* process 1*).*
- *oe1 = (1, 0, 1, "o")*: event *oe1* is the first output of value 0 from *process 1.*
- *oe2 = (1, 1, 1, "o")*: event *oe2* is the first output of value 1 from *process 1.*

- *oe3 = (2, 0, 1, "o")*: event *oe3* is the first output of value 0 from *process 2.*
- *oe4 = (2, 1, 1, "o")*: event *oe4* is the first output of value 1 from *process 2.*
- $C = \{(oe1, ie2), (oe3, ie4), (ie1, ie3), (ie3, re1)\}$
- $O = \neg (((oe1, oe3) \wedge (oe3, oe2)) \vee ((oe3, oe1) \wedge (oe1, oe4)))$

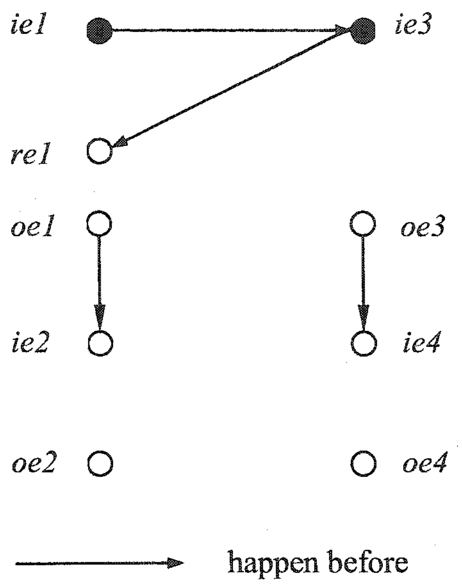


Figure 3.2: A test scenario in the on-line conference example

Figure 3.2 illustrates the graphical representation of the i/o events, remote call events, and the intended control over the timing of their occurrences as described in path constraint *C*. Here, (*oe*1, *ie*2) and (*oe*3, *ie*4) ⊆ *C* expresses the local i/o sequence for process 1 and process 2 respectively. In the test scenario, we require that the first output of value 0 (*oe*1) must happen before the first input of value 1 (*ie*2) in process 1; and correspondingly, the first output of value 0 (*oe*3) must happen before the first input of value 1 (*ie*4) in process 2. Actually, we also have constraints (*ie*1, *oe*1), (*ie*2, *oe*2), (*ie*3, *oe*3), (*ie*4, *oe*4). These constraints are naturally satisfied by the application implementation itself during execution, so we do not need to explicitly express them as part of the constraint. (*ie*1, *ie*3), (*ie*3, *re*1) ⊆ *C* expresses the ordering of the execution across the process boundary: the client of process 1 will send out the signal of *willing to speak* before the client of process 2 does so, but process 1 will not be able to send its ticket number to process 2 (so that the local number of process 2 remains unchanged) until process 2 also picks up its number. As initially the local numbers are all the same, this guarantees that process 1 and 2 will pick up the same number. Finally, the test oracle expressed in *O* essentially says that process 1 and 2 should not be in the critical section simultaneously, i.e. two processes cannot speak at the same time.

The control algorithm adopted in our approach maintains the same level of fairness as the original algorithm implemented in an AUT (e.g., the Distributed Bakery Algorithm in our research). In fact, our testing environment may be used to detect both the fairness and faults of an AUT by analyzing the test scenarios and test results. For instance, if the given test scenario (a test scenario is said to be valid for a program *P* if it is consistent with the specification of this program) is valid but unfair, and the AUT can terminate normally after execute this test scenario, it indicates that there exists unfairness in this AUT. On the other hand, if a given test scenario is feasible to the AUT (meaning that it can be executed by the implementation of the program *P* without causing deadlock or abnormal termination), but this AUT returns an incorrect output, it denotes that we detect a fault in the AUT. In the following, we present the testing environment realizing the above-mentioned control over the execution.

# 4. Java RMI with Interception

Distributed systems require that processes running in different address spaces, potentially on different hosts, be able to communicate with each other [17]. RMI is a distributed object model for the Java programming language that makes distributed processes easy to communicate by means of remote method invocations on distributed objects. RMI allows programmers to develop distributed Java programs with the same syntax and semantics as those that are used for non-distributed programs. It offers a middleware (similar to the CORBA ORB) by which distributed processes can communicate and pass information back and forth.

As mentioned in the introduction, we realize the control of the execution of AUT by modifying the underlying Java middleware layer rather than the AUT implementation. As Java RMI does not provide the interception mechanism, we first insert such a mechanism into the RMI implementation. The RMI implementation is originally built from three abstract layers, i.e., Stub & Skeleton Layer (SSL, for simplicity), Remote Reference Layer (RRL, for simplicity) and Transport Layer [11, 17], as shown in figure 3.1.

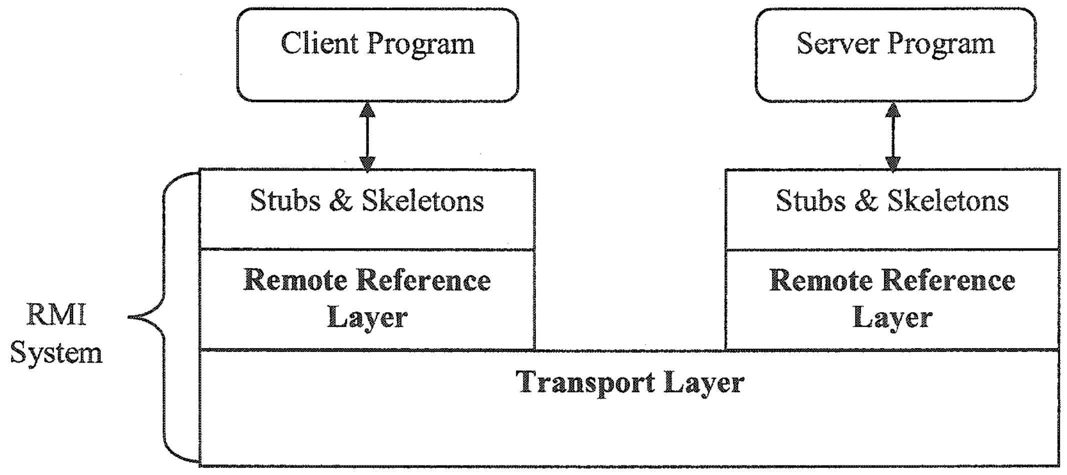


Figure 4.1: Java RMI Architecture Layers [11]

The Stubs and Skeletons layer lies just beneath the view of the developer. This layer intercepts method calls made by the client to the remote interface and redirects these calls to a remote service object. The Remote Reference layer understands how to interpret and manage references made from clients to the remote service objects. In JDK 1.1, this layer connects clients to remote service objects that are running and exported on a server. The transport layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies [11].

Theoretically, the interception service can be implemented in four different levels: 1) inserting the interception between the application and SSL by modifying the Java Naming Service and using Java Dynamic Proxy technique [12, 13], 2) building the interception into the SSL by altering the way that stubs and skeletons are generated, 3) inserting the interception into the RRL by modifying and extending current Java Runtime API, which are class libraries for the Java Runtime environment, and 4) implementing the interception services into the Transport Layer by modifying the existing communication protocols defined by RMI.

## 4.1 Interception Service in Java RMI using Dynamic Proxy

In Java 1.3 software, Sun introduced the Dynamic Proxy class, which is a class that implements a list of interfaces specified at runtime such that a method invocation through one of the interfaces on a proxy instance (an object of the dynamic proxy class) will be encoded and dispatched to another object through a uniform interface. A proxy forces object method calls to occur indirectly through the proxy object, which acts as a delegate for the underlying object being delegated. Proxy objects are usually declared so that the client objects have no indication that they have a proxy object instance. Each proxy instance has an associated invocation handler. When a method is invoked on a proxy instance, the method invocation is encoded and dispatched to the *invoke* method of its invocation handler.

The Dynamic Proxy technique can be viewed as a hook-up mechanism and can be used as a means of the interception service together with some additional interceptor interfaces (please see figure 3.7 for the definitions of interceptors). The *java.lang.reflect.InvocationHandler* is an interface that should be implemented by an interception service to hook users' additional code during a normal execution of a request/reply. However, interception services implemented using dynamic proxy technique can only be used in the client side. This is because the current Java language specification does not have stub class definition for a class implementing *java.lang.reflect.InvocationHandler*, which means that a proxy instance with which this invocation handler associated cannot be a remote object, and thus cannot be transmitted to a remote process or host. This limitation does not allow a client to forward a request to a dynamic proxy object whose implementation is located at the server side. In addition, the dynamic proxy technique for interception services can only be used for the looked-up objects, because we modified the way that Java Naming service works to achieve interception transparency. Client side proxy instances are automatically downloaded to the client process when the client calls the *java.rmi.Naming.lookup* method to retrieve remote objects.

## 4.2 Java RMI Interception Service in the RRL

In the future research, we will adopt the third method to insert interception services into the RRL. We choose not to implement the interception services into the SSL because in the Java 2 SDK, an additional stub protocol was introduced that eliminates the need for skeletons in Java 2 platform-only (and JDK 1.1 compatible) environments. Moreover, injecting interception services into RRL has many advantages over others: 1) it is easier to implement than injecting interceptions in SSL. This is because we will modify some undocumented (also unpublished) Java source files in the Java Runtime libraries (which can be downloaded from Sun's website free of charge). The lack of documents poses many difficulties in understanding the ` ehaviour, workflows, and relationships among the classes in the underlying Java Run-time; 2) inserting interception services into the

RRL gives users the flexibility to introduce or cancel the interception mechanism easily without affecting original Java Run-time API, and 3) with the interception services in Remote Reference Layer, any remote invocations mediated by the RMI middleware can be intercepted. Figure 3.2 illustrates the resulting RMI architecture.

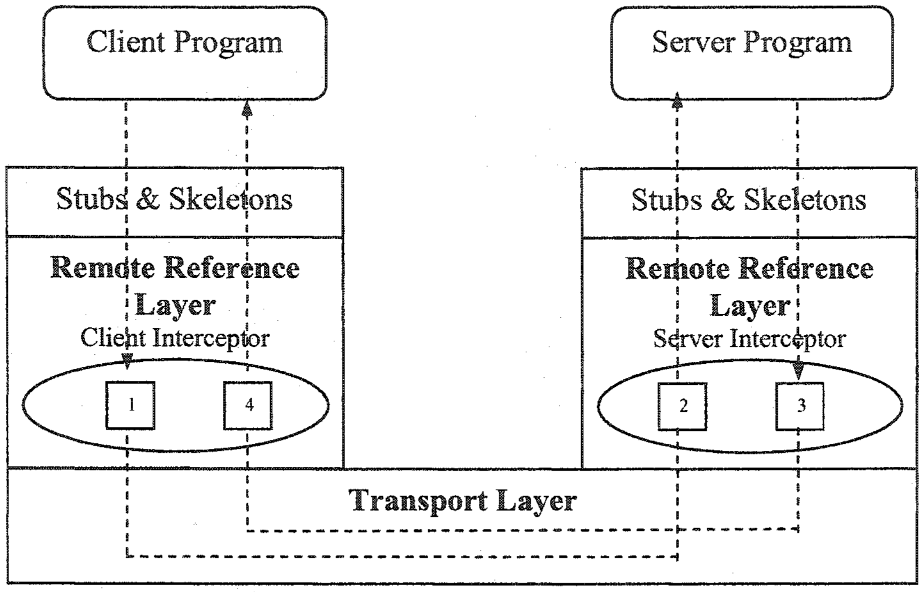


Figure 4.2: Layered architecture in RMI with interception service

We provide *interception services* at both client and server sides, together with some *interception interfaces* that allow users to hook up the testing control mechanisms into the middleware. In figure 3.2, two interception points are defined within each interceptor (*send_request* and *receive_reply* in the RMIClientInterceptor, and *receive_request* and *send_reply* in the RMIServerInterceptor), which are called respectively according to the following order during a request/response processing:

1. The client sends a request, which is caught at the *send_request* (point 1) at the client side;
2. The request is forwarded to the server side and is intercepted at the *receive_request* (point 2);
3. This request is forwarded to the server object for some processing and the response is intercepted at *send_reply* (point 3) before it is sent back to the client.

4. After arriving at the client side, the response from server is first caught by the *receive_reply* (point 4), after which the response is forwarded to the caller.

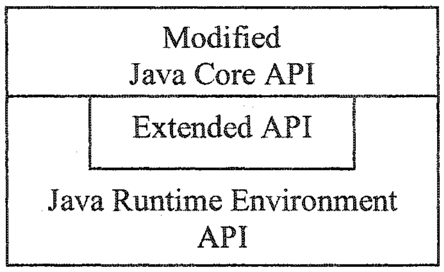


Figure 4.3: High level structure of the modified Java SDK

In order to add this interception service into the RRL, we need to modify the existing Java core API. Figure 4.3 shows a high level structure of the modified Java SDK. In this figure, the modified files of the Java core API are the UnicastRef.java and UnicastServerRef.java. The *UnicastRef* represents the handler for a remote object and will be passed to the client program together with a stub file. A stub uses the *UnicastRef* to carry out a remote method invocation to a remote object. The *UnicastServerRef* represents a server side handler for a remote object and implements the remote reference layer server-side behaviors for remote objects. Both of these files will make use of the Extended API and the Java Runtime Environment API. The Extended API (see figure 4.7 for details) here consists of packages of class files that will be packaged into the JDK class library: the API for RMI interceptors and the interfaces for the testing components. The UML class diagram for the UnicastRef and UnicastServerRef is illustrated in figure 4.4.

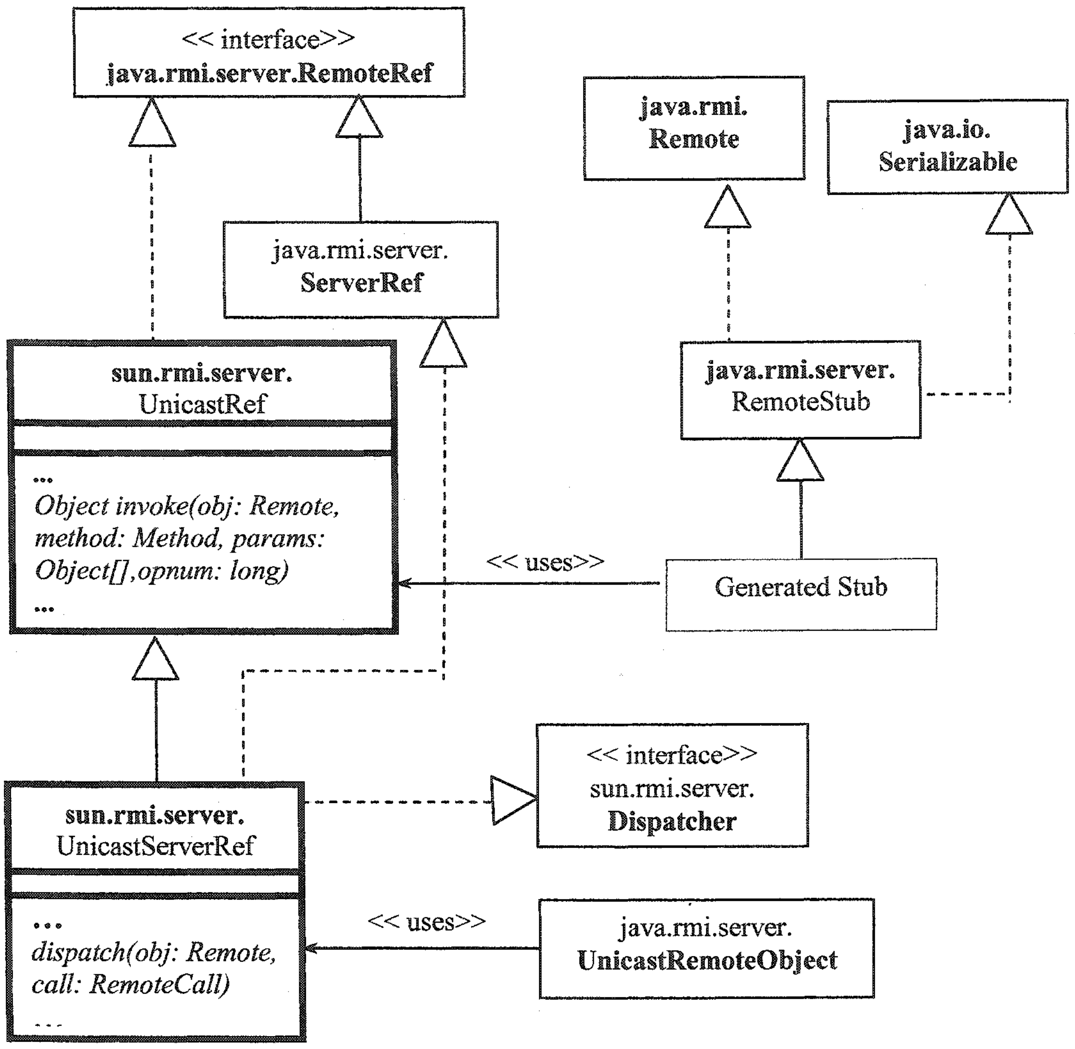


Figure 4.4: The stub class and the UnicastServerRef class in Java core API

As shown in figure 4.2, we basically provide two types of RMI interceptors: RMIClientInterceptor and RMIServerInterceptor. Instances of the RMIClientInterceptor will be downloaded into the client side RRL (instance of the UnicastRef) while instances of the RMIServerInterceptor will be downloaded into the server side RRL (instance of the UnicastServerRef). The code insertion into the RRL (adding code into the UnicastRef and UnicastServerRef) is done before compiling a program. This code injection is done only once for all application under tests and will be packaged into the Java class library.

In the class UnicastServerRef, we are particularly interested in the method *dispatch,* because this is the place where remote method invocation is forwarded to the remote object implementation at the server side. So, in the *dispatch,* we add two statements: *interceptor.receive_request* and *interceptor.send_reply* right before and after the real method invocation on the remote object. Thus, the flow of execution of a remote call will be captured at the *receive_request* and *send_reply* interception points (at the callee's side), which will in turn exercise the code implemented at these two points. The pseudo code for the modified class is given below:

```
import ca.uwindsor.kunwang.rmi.interceptor.*;
// import other packages;

public class UnicastServerRef extends UnicastRef implements ServerRef, Dispatcher {
        private RMIServerInterceptor[] interceptors;
        private RMIInterceptorServer iserver;
        private ServerRequestInfo re = new ServerRequestInfo ();

        ... // Other variable definitions of this class
        public UnicastServerRef() {
                try {
                   if (iserver = = null)
                                iserver = (RMIInterceptorServer) java.rmi.Naming.lookup
                                                ("rmi://localhost/InterceptorServer");
                } catch (Exception e) {
                        System.err.println("Obtaining Iserver exception: "
                                                + e.getMessage());
                        e.printStackTrace();
                }
        }

        ... // Other constructors and methods of this class

        public void dispatch(Remote obj, RemoteCall call) throws IOException {
                ... // Other part of this method
                Class[] interfaces = obj.getClass().getInterfaces();
                String interfacename = interfaces[0].getName();
                if (!interfacename.equals("ca.uwindsor.kunwang.rmi.interceptor.
                                                        RMIInterceptorServer")) {
                        if (iserver != null && interceptors == null)
                                interceptors = iserver.getServerInterceptors();
                        re.setIname(interfacename);
                        re.setMname(method.getName());
                re.setClientRef(getClientRef());
                re.setClientHost(getClientHost());
            }

            try {
                if (interceptors != null) {
                    for (int i = 0; i < interceptors.length; i++)
                        interceptors[i].receive_request(re);
                }

                result = method.invoke(obj, params); // Real Method Invocation

                if (interceptors != null) {
                    for (int i = 0; i < interceptors.length; i++)
                        interceptors[i].send_reply(re);
                }
            } catch (InvocationTargetException e) {
                    throw e.getTargetException();
            }
            ... // Other part of this method
        }
        ... // Other methods of this class
}
```

Figure 4.5: Pseudo-code of the Modified UnicastServerRef class

Before we give details of the above modified *UnicastServerRef,* and the *UnicastRef* that will be introduced soon, let us briefly explain the Java Reflection API because it plays a very important role in providing run-time information of Java objects, their running environment and their interactions.

The Java Reflection is a built-in API in Java language, which represents, or reflects, the classes, interfaces, and objects in the current Java Virtual Machine. This reflection API is often used when writing development tools such as debuggers, class browsers, and GUI builders. With the reflection API you can do things such as [27]:

- Dynamically determine the class of an object
- Get information about a class's modifiers, fields, methods, constructors, and super classes or implemented interfaces

- Find out what constants and method declarations belong to an interface
- Create an instance of a class whose name is not known until runtime
- Invoke a method on an object, even if the method is not known until runtime

Now, let us look at the above code in more detail (the newly added code is in italic and bold font). In the current implementation of Java RMI, an instance of the UnicastServerRef is created whenever a remote object is exported either implicitly (by extending UnicastRemoteObject) or explicitly through the *exportObject* method of the *UnicastRemoteObject* class. A remote object is not ready to receive requests until it is exported. For each instance of the UnicastServerRef, instances of the RMIServerInterceptor are downloaded to the process where the remote object is defined. This is possible because the user-defined interceptors are implementations of *java.io.Serializable.*

In the classes UnicastServerRef and the modified UnicastRef that we will introduce below, the interceptors are looked up on a per-request basis. This means that request at both client and server sides will check if interceptors have been registered into the Interceptor Server, whose reference can be retrieved from the RMI Naming service (rmiregistry) on local host. The Interceptor Server, denoted as *iserver* is a remote object providing interceptor registration and lookup services in each host and is registered in a host using a reserved name "rmi://localhost/InterceptorServer", when the testing environment is started up. For simplicity, we assume that there is only one process running on each host with one interceptor server for each process, in this testing environment. The reason we make this assumption here is that the code for looking up the Interceptor Server (i.e.: *iserver = (RMIInterceptorServer) java.rmi.Naming.lookup("rmi://localhost/InterceptorServer");)* is generic to the Java Runtime Environment in a machine and assumes only one Interceptor Server (through the name *localhost/InterceptorServer*) on this machine. Therefore, such a solution to the interception service in the middleware is not able to handle situations when multiple processes are running on the same host.

Now, because the interceptors are looked up on a per-requests basis, we get one or more interceptors for each remote object on the server side. Whenever a method invocation is made remotely, this invocation will be directed to the *dispatch* method at the server side, which in turn forwards this invocation to the object implementation via the statement *method.invoke*. The additional code (user implemented) is executed before and after this statement by making a series of calls to the registered interceptors. The *method* is an instance of the class Method in the Java Reflection, which provides information about, and access to, a single method on a class or interface. The statement *method.invoke(obj, params)* takes an object (*obj*) and an array of objects (*params*) as parameters and invokes the underlying method represented by this Method object, on the specified object (*obj*) with the specified parameters (*params*).

As far as the testing control concerns, the information we are interested in within the *dispatch* method are the interface name and the method name of a remote object that are invoked remotely. Here, we assume that the remote object being called only implements one remote interface, and this information can also be obtained via Java Reflection: *obj.getClass().getInterfaces()*. In fact, a remote object may implement more than one remote interface in a real application. In such a case, the interface name cannot be obtained simply by calling *interfaces[0].getName()* because the invoked method may be defined in some other interfaces (e.g.: *interfaces[1]*), and this information cannot be known until run-time. Again, this problem can be solved by the Java Reflection. By using Java Reflection, we can compare at run-time, the name and parameters of the invoked method with those of the public methods defined in all implemented remote interfaces. If there is one method matching that of the invoked method, then the interface that defines that specific method is the one that we are looking for. However, this solution is based on another assumption: there cannot be identical public method definitions in those implemented interfaces. In our research, we just choose the first assumption for simplicity.

After the interface and method names are acquired, they are encapsulated into a *ServerRequestInfo* object (see class definition in figure 4.7), which is then passed as a parameter into the *receive_request* and *send_reply* methods. The *ServerRequestInfo* object is also part of the extended API and provides some information about this remote event (just like that of the *ServerRequestInfo* in the CORBA core specification [16]). This information will be used for test control purposes. We will explain how the testing components use this information in Chapter 5.

Very similarly, we add two interception points (*send_request* and *receive_reply*) in the UnicastRef, right before and after this object carries out a remote method invocation. The modified class looks like this:

```
import ca.uwindsor.kunwang.rmi.interceptor.*;
// import other packages;
public class UnicastRef implements RemoteRef {

        private RMIClientInterceptor[] interceptors;
        private RMIInterceptorServer iserver;
        private ClientRequestInfo re = new ClientRequestInfo();

        ... // Other part of this class

        public Object invoke(Remote obj, java.lang.reflect.Method method,
                                Object[] params, long opnum) throws Exception {
                Class[] interfaces = obj.getClass().getInterfaces();
                String interfacename = interfaces[0].getName();

                if (!interfacename.equals("ca.uwindsor.kunwang.rmi.interceptor.
                                                RMIInterceptorServer")) {
                        try {
                           if (iserver == null)
                                iserver = (RMIInterceptorServer)java.rmi.Naming.lookup
                                        ("rmi://localhost/InterceptorServer");
                        } catch (Exception e) {
                        }
                        re.setIname(interfacename);
                        re.setMname(method.getName());
                        re.setTarget(obj);

                        if (iserver != null && interceptors == null)
                                interceptors = iserver.getClientInterceptors();
                }

                if (interceptors != null) {
                                for (int i = 0; i < interceptors.length; i++)
                                        interceptors[i].send_request(re);
                }

                // Real Method Invocation
                call.executeCall();

                if (interceptors != null) {
                                for (int i = 0; i < interceptors.length; i++)
                                        interceptors[i].receive_reply(re);
                }

                ... // other part of this method
        }

        ... // other part of this class
}
```

Figure 4.6: Pseudo-code of the Modified UnicastRef class

As mentioned previously, the code in italic and bold font is injected into the UnicastRef before compiling a program. The UnicastRef is an instance of the *RemoteRef*, which represents the handler of a remote object. A RemoteStub (e.g., a stub class) uses an instance of the UnicastRef to carry out a remote method invocation to a remote object. This *invoke* method takes as parameters the remote object reference being called upon, the method to be invoked, the parameter list and a hash that may be used to represent the method, and returns the result of the remote method invocation. In the *invoke* method, the real method invocation is carried out by the statement: *call.executeCall.* The interceptor downloading mechanism in *UnicastRef* is the same as that of *UnicastServerRef*, so we also get one or more instances of the RMIClientInterceptor for each instance of *UnicastRef* at the client side.

In figure 4.3, we also showed that the modified Java core API utilizes our Extended API, which basically contains ten class files, namely *RMIClientInterceptor,* *RMIServerInterceptor*, *RMIInterceptorServer*, *RequestInfo*, *RequestInfoImpl*, *ClientRequestInfo*, *ServerRequestInfo*, *ClientRequestInfoImpl*, *ServerRequestInfoImpl* and the *InterceptorInitializer*. All user-defined interceptors must extend the *RMIClientInterceptor* and/or *RMIServerInterceptor* either directly or indirectly. In order to become part of the RMI implementation, user-defined interceptor instances must be registered into the *RMIInterceptorServer* in either of the following two ways:

*1)* By registering an associated RMI interceptor initializer, which implements the *InterceptorInitializer* interface.

*2)* By writing a program, in which the reference to the *InterceptorServer* must be obtained and the methods *addClientInterceptor* and/or *addServerInterceptor* must be called explicitly.

The ten classes described above provide users a "standard" method to create their own interceptors, to register them into the middleware and to obtain information about a remote event. We say this method is "standard" because it allows users to inject their own code into the RMI middleware by implementing and extending a set of CORBA-like classes, and starting this interception service in a way very similar to that of CORBA. The static relationships among these interception interfaces is given in figure 4.7:

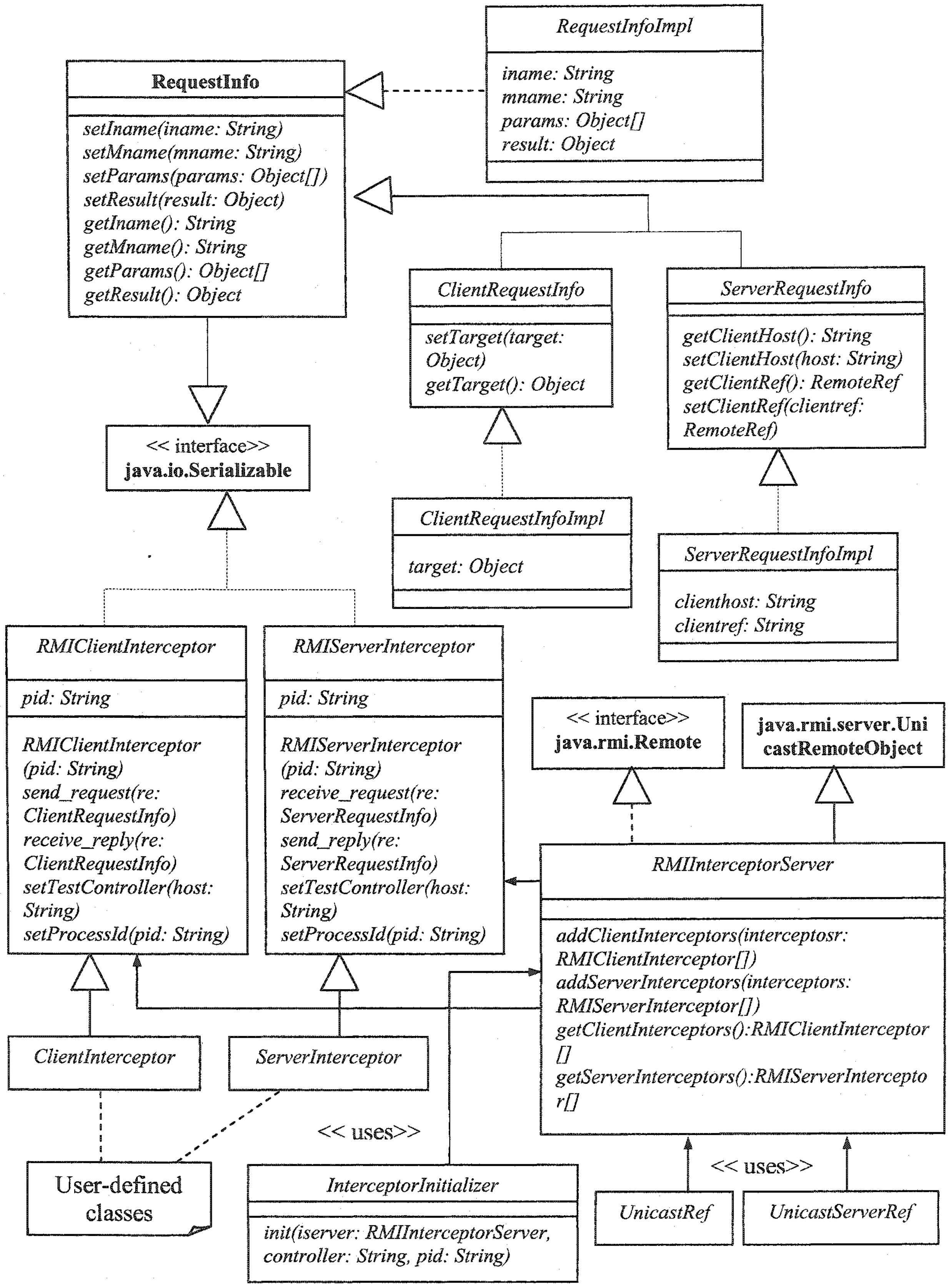

RequestInfo
setIname(iname: String)
setMname(mname: String)
setParams(params: Object[])
setResult(result: Object)
getIname(): String
getMname(): String
getParams(): Object[]
getResult(): Object
RequestInfoImpl
iname: String
mname: String
params: Object[]
result: Object
ClientRequestInfo
setTarget(target: Object)
getTarget(): Object
ServerRequestInfo
getClientHost(): String
setClientHost(host: String)
getClientRef(): RemoteRef
setClientRef(clientref: RemoteRef)
<< interface>> java.io.Serializable
ClientRequestInfoImpl
target: Object
ServerRequestInfoImpl
clienthost: String
clientref: String
RMIClientInterceptor
pid: String
RMIClientInterceptor (pid: String)
send_request(re: ClientRequestInfo)
receive_reply(re: ClientRequestInfo)
setTestController(host: String)
setProcessId(pid: String)
RMIServerInterceptor
pid: String
RMIServerInterceptor (pid: String)
receive_request(re: ServerRequestInfo)
send_reply(re: ServerRequestInfo)
setTestController(host: String)
setProcessId(pid: String)
<< interface>> java.rmi.Remote
java.rmi.server.UnicastRemoteObject
RMIInterceptorServer
addClientInterceptors(interceptosr: RMIClientInterceptor[])
addServerInterceptors(interceptors: RMIServerInterceptor[])
getClientInterceptors():RMIClientInterceptor []
getServerInterceptors():RMIServerInterceptor[]
ClientInterceptor
ServerInterceptor
<< uses>>
User-defined classes
InterceptorInitializer
init(iserver: RMIInterceptorServer, controller: String, pid: String)
<< uses>>
UnicastRef
UnicastServerRef

Figure 4.7: Static Relationship for the Interception Service Implementation

More precisely, a user who wants to build interceptors into the RMI middleware by means of the first method should follow the three steps:

1. Write his own interceptor implementation by extending *RMIClientInterceptor* and/or *RMIServerInterceptor*
2. Implement the *InterceptorInitializer* interface and registers the above interceptors by calling the *addClientInterceptors* and/or *addServerInterceptors* on the *RMIInterceptorServer*, whose reference is passed into the *InterceptorInitializer* as a parameter when it starts up. The location of the user-implemented *InterceptorInitializer* class is specified in a batch file, which is a command file used to start the *RMIInterceptorServer.*
3. Start up the *RMIInterceptorServer* provided by our interception service.

This is a simple way to register user-defined interceptors when the testing environment starts up. In the case that a user would register interceptors by means of the second method (dynamically register interceptors), one must start up the *RMIInterceptorServer* before running the program containing the code of registering interceptors. This is a more complex yet flexible way to register user-defined interceptors, which gives users the flexibility to decide when to register or unregister the interceptors on the fly. A sample code for this approach is given below:

```
Create an array of RMIClientInterceptor cinterceptors;
Create an array of RMIServerInterceptor sinterceptors;
try {
        if (iserver = = null)
                iserver = (RMIInterceptorServer)java.rmi.Naming.lookup
                                ("rmi://localhost/InterceptorServer");
        else {
                // Add client interceptors
                iserver.addClientInterceptors(cinterceptors);
                // Add server interceptors
                iserver.addServerInterceptors(sinterceptors);
        }

} catch (Exception e) {
        System.err.println("Obtaining Iserver exception: "
                                + e.getMessage());
                e.printStackTrace();
        }
```

Fig. 4.8: Sample code for manually register interceptors

No matter which way to use, the *UnicastRef* and *UnicastServerRef* will download those interceptors (via *getClientInterceptors* and *getServerInterceptors*) as local objects whenever allowed.

In our testing architecture, the interceptor implementation will make a series of calls to the local test controller in order to hook up the testing control mechanism. However, the built-in interception service itself is independent of any testing tool.

# 5. Testing Architecture

Now with the help of the interception service in the middleware layer, we are able to incorporate a testing environment to control the remote calls without modification of the AUT. Our testing environment uses this facility to realize the automated control over the input and the remote call events. Figure 5.1 illustrates its architecture when the AUT (i.e., online conference) consists of two processes running on two hosts and communicating with each other via Java RMI. In this figure, we only show method calls in one direction (from process 1 to process 2); the control flow of method calls from the other direction is analogous. We also omit the Stubs & Skeletons Layer and the Transport Layer since the only layer we are interested in here is the RRL, in which we can illustrate how the testing components work together to control the testing process. In this architecture, process 1 holds a remote object *obj*2 defined in process 2 and process 2 holds a remote object *obj*1 defined in process1. With this setting, whenever a remote call is made on an object *obj*, it is caught both at the *send_request* at the caller's side and at the *receive_request* at the callee's side. Analogously, when this call is returned, it is caught both at the *send_reply* at the callee's side and at the *receive_reply* at the caller's side. These four control points correspond to the four types (i.e. *qc, qs, ps, pc*) of remote call events defined in the test scenario. Because the AUT processes communicate with each other in a peer-to-peer manner, we register both the *RMIClientInterceptor* and the *RMIServerInterceptor* in each Java Virtual Machine to capture method invocations on both remote objects.

The testing environment is distributed: each process under test (PUT) has a local path controller. During the lifecycle of a request/reply, the client and server interceptors inform the local path controllers of the request or response that they catch and will let the execution of AUT continue only with the permission from the path controllers. These path controllers contain the same path constraint information contained in the test scenario, and the current global states of the running AUT (i.e. which events have already happened). Each PUT also has a local test driver, which reads the test case file and is in charge of providing local input to it *at an appropriate time*. The local test drivers make their decision on *when* to provide input to the process based on the permission from local path controllers. After an i/o event or remote event happens, both the test driver and interceptors will notify the local path controller to update the global states of the running AUT. This controller then informs the update to the communicator, which in turn broadcasts this information to all other path controllers.

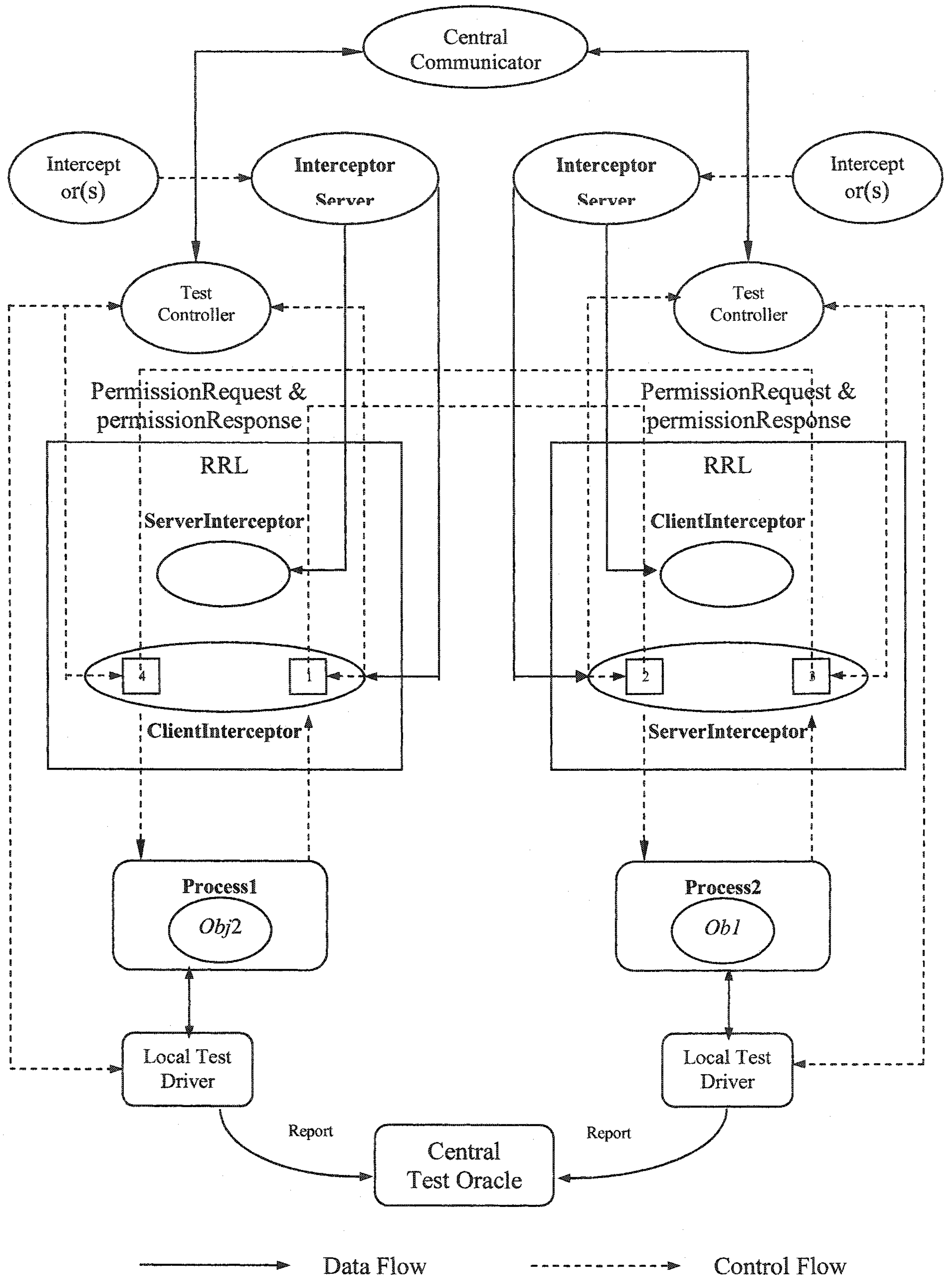
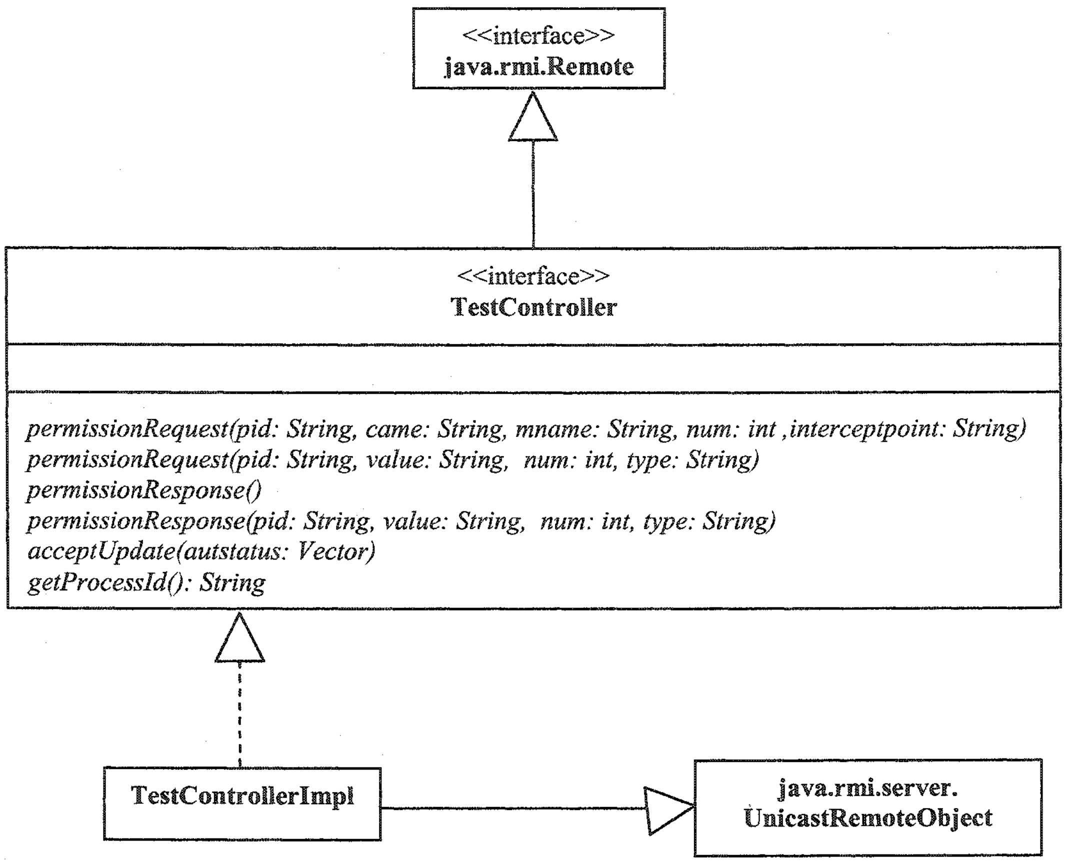


Figure 5.1: The distributed testing environment with automated control

The condition $O$ to be checked on the output is kept in a centralized test oracle. In a general sense, a *test oracle* is means of checking test results against expected results. The central test oracle in our work is a program that will receive an output (test results) from each local test driver, determine if a test passes or fails with respect to the given test oracle (expected results) derived from the requirement specification, and reports the errors whenever encountered.

In the above on-line conference example, there is one remote object in each process. The path constraint ($C = \{(oe1, ie2), (oe3, ie4), (ie1, ie3), (ie3, re1)\}$), all i/o events and remote events are initially given to the two path controllers. A copy of the i/o events are also given to the two local test drivers. The test driver of process 2 initially asks for permission from its controller, and is blocked because of condition ($ie1$, $ie3$). The test driver of process 1 is granted the permission to give input $ie1$ to process 1 and update the controller's status. The controller will broadcast this update to all other controllers via the central communicator and wake up those processes blocked on it. In particular, when the test controller of process 2 receives this message, it enables the test driver to feed input of $ie3$. However, if process 1 proceeds to make a remote call *permissionRequest* ($re1$) before $ie3$ happens, this call is first caught by an instance of the RMIClientInterceptor, which informs the path controller of process 1. Since $(ie3, re1) \in C$, the test controller will not allow $re1$ to happen until input $ie3$ is given to process 2. By this control, we guarantee that both processes locally pick up the same number.

During a testing procedure described above, the test controller plays a key role in deciding whether or not to allow an event to happen. Like the *RMI Interceptors,* all test controllers must either implement the TestController interface or extend a subclass of type TestController, which is also part of the Extended API. The class diagram of the TestController is shown in figure 5.2:



<<interface>>
java.rmi.Remote
<<interface>>
TestController
permissionRequest(pid: String, came: String, mname: String, num: int ,interceptpoint: String)
permissionRequest(pid: String, value: String, num: int, type: String)
permissionResponse()
permissionResponse(pid: String, value: String, num: int, type: String)
acceptUpdate(autstatus: Vector)
getProcessId(): String
TestControllerImpl
java.rmi.server.
UnicastRemoteObject

Figure 5.2: Class diagram of the TestController

In the interface *TestController,* we define two *permissionRequest* methods, one for the remote events and the other for the input events. The signatures of these methods are based on the definitions in section 2.3, where input events and remote events have different formats, which will be used to identify a specific event by the controller. We also define two *permissionResponse* methods. The first *permissionResponse* is called by a test driver or an interceptor, after an input or remote event successfully completes. Because we do not place any constraint over the orders of output events, but still need to notify other controllers that a specific output event has happened, we call the second *permissionResponse* method after an output event has happened.

## 5.1 Control of Remote Events – Implementing Interceptors

For the control of the orders of remote events, we must be able to call the *permissionRequest* method on the TestController at certain interception points (i.e., "*qc*", "*qs*", "*pc*", and "*ps*") to get permission before a process is allowed to proceed, and then to call the *permissionResponse* method after this event happens. This can be implemented in the user-defined interceptors. The code of a client side interceptor example, *ClientInterceptor* is given bellow:

```
import ca.uwindsor.kunwang.rmi.interceptor.*;
public class ClientInterceptor extends RMIClientInterceptor {

        private TestController tc;
        private String controllerhost;
        private String pid;
        private int qcnum = 0; // number of invocation
        private int lastdot;
        private String iname; // Name of the called interface without package name
        private Class calledClass; // Class object of the called class
        private Class tcInterface; // Class object of the test controller

        ...// Other variable definitions

        public ClientInterceptor(String pid) {
                super(pid);
        }

        public void send_request(RemoteEvent re) {
                try
                        /* check if the called class is an instance of the TestController, and
                        if it is, the test controller will not be retrieved.
                        */

                        calledClass = Class.forName(re.getIname());
                        tcInterface = Class.forName
                                ("ca.uwindsor.kunwang.rmi.testing.TestController");
                        lastdot = re.getIname().lastIndexOf('.');
                        iname = re.getIname().substring(lastdot + 1);

                        if (tc = = null && !tcInterface.isAssignableFrom(calledClass)) {
                                tc = (TestController)java.rmi.Naming.lookup
                                                        ("rmi://localhost/TestController");
```

```
                    }

                    if (!iname.equals("TestController") && re.getMname().equals
                                                        ("permissionRequest")) {
                              qcnum ++;
                              tc.permissionRequest(pid, iname, re.getMname(), qcnum,
                                                                            "qc");
                              tc.permissionResponse();
                    }
          } catch (Exception e) {
                    System.err.println("Obtaining IServer exception: "
                              + e.getMessage());
                    e.printStackTrace();

          }
     }

     public void receive_reply(String iname, String mname) {
          // We do not need to implement this interception point according to the
          // test scenario specification. We only need to implement control at one
          // interception point: "qc".
     }

     public void setTestController(String controllerhost) {
          this.controllerhost = controllerhost;
     }

     public void setProcessId(String processid) {
          pid = processid;
     }
}
```

Figure 5.3: Code for an example of client side interceptor

The class *ClientInterceptor* has a constructor taking the *pid* as its parameter. This parameter represents the id for the process to which this *ClientInterceptor* will delegate requests, and is passed into the constructor by the interceptor server when the server is started. In the method *send_request,* we first determine if the called class is an instance of the interface TestController. If it is an instance of the TestController, or the current test controller (tc) is not null, we do not need to make a remote call to retrieve the test controller.

In an interceptor's implementation, if an interceptor itself makes a remote invocation, it shall have some means of breaking infinite recursion. For example: the client calls the method *X* on a remote object; this call is captured at the client-side stub and the *send_request* is called, which makes remote invocation *permissionRequest* or *permissionResponse* on the test controller; so *send_request* is called, which again calls method *permissionRequest* or *permissionResponse*; and so on unless the implementation of *send_request* breaks the recursion. In the *ClientInterceptor*, the second if statement: *if (iname.equals("OnlineConference") && re.getMname().equals("permissionRequest"))* is very important because it avoids the remote call recursion and unnecessary remote calls to the test controller.

## 5.2 Control of Input Events – Test Drivers

Apart from enforcing some constraints over the orders of remote events in the RMI Interceptors, we also enforce constraints over the orders of input events. This is realized in the test drivers. The control of input events is quite straight forward: we only need to request permission from the Test Controller by calling the *permissionRequest* method before the driver feeds input to its process (input event) and then send a response to the Test Controller by calling the *permissionResponse* method after this input event happens. We will not discuss the code of a test driver in detail since the syntax for making these requests and responses in a test driver is very similar to those in a RMI interceptor and they are different only in parameter formats.

## 5.3 Control Algorithm

In previous parts of this chapter, we have discussed how the testing components work together to realize the control over the orders of occurrences of input events and remote events. We also talked about in detail how this control is performed in RMI interceptors and test drivers respectively. Now, we will discuss how this control is implemented in the Test Controller. The algorithm for a test controller, *TestController* is given bellow:

*public class TestControllerImpl extends UnicastRemoteObject*

```
                implements TestController {

Definitions of instance variables: pointers, tests, constraints and communicator;

public synchronized void acceptUpdate(Vector autstatus) throws RemoteException {
        pointers = autstatus;
        this.notifyAll();
}

public synchronized void permissionRequest(String processid, String classname, String mname, int
                                num, String interceptpoint) {
        permissionRequest(processid, classname, mname, num, interceptpoint, constraints);

}

private void permissionRequest(String processid, String classname, String   mname, int num,
                String interceptpoint, LinkedList constraints) {

        boolean flag = false; // Signal whether a specific event has happened.

        // Record the updated states after a certain event happens.
        updatedstates = new Hashtable();
        try {
                Decide whether this reqEvent is one specified in the test scenario;

                if (reqEvent != null) {
                        while (true) {
                                // To check if a certain event is contained in the test constraint.
                                boolean inConstraint = false;
                                for (int i = 0; i < constraints.size(); i++) {
                                        Vector aconstraint = (Vector)constraints.get(i);
                                        for (int j = 0; j < aconstraint.size(); j++) {
                                                if reqEvent matches a specific event in the
                                                element {
                                                        pos = pointers.get(i);
                                                        if (j > pos)
                                                        System.out.println(eventname + " is
                                                                        blocked here!");
                                                        wait();
                                                } else {
                                                        updatedconstraints.put(new
                                                        Integer(i), new Integer(j + 1));
                                                        flag = true;
                                                }
                                                break;

                                        }
                                }

                                // A certain event is not contained in the test constraint.

                                if (!inConstraint && i == constraints.size() - 1) {
                                        if (happentime == 0) {
                                                rightNow = new Date();
                                                happentime = rightNow.getTime();
                                        }
                                        flag = true;
                                   }
                              }

                              // If the reqEvent has happened, break from the loop; otherwise,
                              // try to get permission again.
                              if (flag)
                                   break;
                         }

                    System.out.println(eventname + " has happened!");
               }
          } catch (Exception e) {
                    e.printStackTrace();
          }
     }

     public void permissionResponse() {
          try {
                    communicator.updateStatus(updatedconstates);
          } catch (Exception e) {
                    e.printStackTrace();
          }
     }

     public static void main(String[] args) {
          try {
                Initialize the pointers, tests and constraints;
                Retrieve the reference to the communicator;
     } catch (Exception e) {
          System.err.println("Test Controller exception: " + e.getMessage());
          e.printStackTrace();
     }
     }
}
```

Figure 5.4: The algorithm for a test controller

To control the execution orders, this algorithm uses some important data structures to store the test constraint and test cases derived from the test specification file, and record the current status of the running processes. These data structures include:

- *constraints,* which is a Java *LinkedList* to store the test constraint derived from the test specification. The constraint given in our test scenario is $C = \{(oe1, ie2), (oe3, ie4), (ie1, ie3), (ie3, re1)\}$, containing four elements separated by commas. Each element can be assigned a number (0, 1, ..., n), and within each element (($oe1$, $ie2$) for example), the position of an event in this element denotes the specified order of occurrence of this event. In this example, the positions of *oe*1 and *ie*2 are 0 and 1 respectively, so *oe1* should happen before *ie2*.

- *pointers*, which is a Java *Vector* to record the positions of events that should happen next in all elements of the test constraint. The size of the variable *pointers* is the number of elements contained in the test constraint, and the values in the *pointers* are the current positions for each element in the test constraint. A value in *pointers* will increase by one if a specific event happens.

- *tests,* which is a Java *Hashtable* to store the test cases obtained from the test specification. This variable stores the format of each event specified in the test scenario and allows a requesting event to be compared with these events.

Now, whenever a process calls the *permissionRequest* of the *TestController* with parameters, the *TestController* will decide whether this requesting event (*reqEvent*) is specified in the tests. If it is the one that is specified in the test scenario, then for each element in the test constraint, the algorithm retrieves from *pointers*, the position (*pos*) of the event that is the next event to be executed in this element. This algorithm checks if the position (*j*) of *reqEvent* is greater than *pos.* If yes, it shows that certain events that should happen before *reqEvent* have not happened yet. So, this process should be blocked. Otherwise, this process is granted the permission to proceed, and the system states should be updated.

After this event happens, a process calls the *permissionResponse* of the *TestController* to notify the *communicator* of the update of the system status. The *communicator* in turn calls the *acceptUpdate* of every other *TestController* to notify them of this update. This method resets the values of *pointers* to the new values in *autstatus* and wake up all the processes that are waiting on the *TestController.*

## 5.3 Automating Testing Environment Setup

As mentioned in Chapter 3, the testing components in the testing environment are distributed, which allows us to design a scalable testing architecture. The central communicator and the central test oracle can be installed on any dedicated host(s). For performance, this architecture requires that a test driver, a test controller and an RMI interceptor server to be installed on the same host with a process. Although the test controllers are running in the same hosts as their processes, they have to be Java remote objects that work both as clients and servers, and communicate with test drivers and the communicator via Java RMI. Our testing environment is scalable in the sense that the size of an AUT can be larger or smaller. In the example showed in figure 5.1, the AUT only consists of two processes running on two machines; however this testing architecture allows users to handle an AUT consisting of any number of processes and hosts, as long as they comply with those specifications defined in our testing architecture. This scalability is facilitated by allowing users to configure an XML file, i.e., configuration.xml in which they can configure the global settings of the testing environment. This XML file must conform to the Document Type Definition (DTD) defined in the testing environment, in order to be validated and interchanged by independent groups of developers. The DTD file (Configuration.dtd) defined in our testing infrastructure is given below:

```
<?xml version='1.0' encoding='utf-8'?>

<!--
   DTD for the Configuration.xml.
-->

<!ELEMENT TestConfig (Communicator, TestOracle, (InterceptorServer, Controller,
Driver)+)>
<!ELEMENT Communicator (IP, TotalProcess, Source, Policy)>
<!ELEMENT TestOracle (IP, TestCase, Source, Policy)>
<!ELEMENT InterceptorServer (Name, Source, Policy, InterceptorInitializerClass,
                                                                    ProcessID)>
<!ELEMENT Controller (Name, IP, Source, Policy)>
<!ELEMENT Driver (Name, AUTCommand)>
<!ELEMENT IP (#PCDATA)>
```

```
<!ELEMENT TotalProcess (#PCDATA)>
<!ELEMENT Source (#PCDATA)>
<!ELEMENT Policy (#PCDATA)>
<!ELEMENT TestCase (#PCDATA)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT InterceptorInitializerClass (#PCDATA)>
<!ELEMENT ProcessID (#PCDATA)>
<!ELEMENT AUTCommand (#PCDATA)>
```

Figure 5.5: The DTD definition for the Configuration.xml file

Now, let us look at some details of the above DTD. The testing environment denoted as the root element *TestConfig* defines a single *Communicator* and a single *TestOracle*. For each process in an AUT, an *InterceptorServer* element, a *Controller* element and a *Driver* element are required.

The *Communicator* is an ordinary RMI remote object; hence its sub elements should contain *IP*, *Source*, and *Policy*, which are all required system properties when the Communicator is started. These three elements represent the IP address of the host that the communicator will be running on, the location of the source files of the *Serializable* classes that may be downloaded, and the location of the policy file (which specifies the security policy for the Communicator host machine) respectively. The *TotalProcess* element in the *Communicator* represents the number of processes under test. We use it here because when we start the processes one by one, we need to block the progress of a test driver (thus block the progress of its process) until all processes successfully start up and ready to communicate with each other. This number is also used in the *TestOracle* to determine if all processes have terminated successfully.

The *TestOracle* maintains a single copy of the test case file, which is represented by the *TestCase* element. The *Name* element in each parent element is used as the name of the generated batch file. The *InterceptorInitializerClass* element denotes the fully qualified name of the user-implemented InterceptorInitializer class, and will be used as a run-time argument to the Interceptor Server. The *ProcessID* in the *InterceptorServer* will be obtained by interceptors as part of information to make permission requests from the test controller. The *AUTCommand* element tells the test driver where to find the PUT to be started.

A utility is provided to read the testing configuration and generate all the necessary batch files to start the testing components, interception services, and processes. An sample configuration of the Configuration.xml file for the OnlineConference application looks like this:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE TestConfig SYSTEM "Configuration.dtd">

<TestConfig>

        <Communicator>
                <IP>137.207.16.49</IP>
                <TotalProcess>2</TotalProcess>
                <Source>D:\thesis\implementation\</Source>
                <Policy>D:\thesis\implementation\policy.txt</Policy>
        </Communicator>

        <TestOracle>
                <IP>137.207.16.49</IP>
                <TestCase>D:\Thesis\implementation\testcase.txt</TestCase>
                <Source>D:\thesis\implementation\</Source>
                <Policy>D:\thesis\implementation\policy.txt</Policy>
        </TestOracle>

        <InterceptorServer>
                <Name>iserver0</Name>
                <Source>D:\thesis\implementation\</Source>
                <Policy>D:\thesis\implementation\policy.txt</Policy>
                <InterceptorInitializerClass>test.MyInterceptorInitializer
                </InterceptorInitializerClass>
                <ProcessID>Peer-0</ProcessID>
        </InterceptorServer>

        <InterceptorServer>
                <Name>iserver1</Name>
                <Source>D:\thesis\implementation\</Source>
                <Policy>D:\thesis\implementation\policy.txt</Policy>
                <InterceptorInitializerClass>test.MyInterceptorInitializer
                </InterceptorInitializerClass>
                <ProcessID>Peer-1</ProcessID>
        </InterceptorServer>


        <Controller>
                <Name>controller0</Name>
                <IP>137.207.16.49</IP>
                <Source>D:\thesis\implementation\</Source>
                <Policy>D:\thesis\implementation\policy.txt</Policy>
        </Controller>

        <Controller>
                <Name>controller1</Name>
                <IP>137.207.234.189</IP>
                <Source>D:\thesis\implementation\</Source>
                <Policy>D:\thesis\implementation\policy.txt</Policy>
        </Controller>

        <Driver>
                <Name>driver0</Name>
                <AUT>D:/Thesis/implementation/Peer0.bat</AUT>
        </Driver>

        <Driver>
                <Name>driver1</Name>
                <AUT>D:/Thesis/implementation/Peer1.bat</AUT>
        </Driver>

</TestConfig>
```

Figure 5.6: A sample Configuration.xml file

By configuring the above file and running the utility, ConfigGenrator, we automatically generate a single copy of batch file for the Communicator and for the TestOracle respectively and a set of batch files for the Interception Server, Test Controller and Test Driver for the AUT. The former two batch files are to be copied to other machine(s) and the latter three batch files together with the process are to be delivered to each individual machine. We also maintain a single copy of the testcase.txt file because it is frequently updated for different test scenarios. Currently, we install the testcase.txt file in the same machine with the TestOracle, through which the test drivers and test controllers will read this file into its local drive each time they start. Here is an example of the generated batch file for TestOracle (testoracle.bat), based on the information in Configuration.xml:

```
java -Djava.rmi.server.codebase=file:/D:\thesis\implementation\
    -Djava.rmi.server.hostname=137.207.16.49 -Djava.security.policy=policy.txt
    TestOracleImpl
    namingserver=rmi://137.207.16.49/
    totalprocess=2
```

Figure 5.7: An example of generated batch file for TestOracle

# 6. Overview of CORBA

## 6.1 A Brief Overview of Common Object Request Broker Architecture

This section reviews some fundamental concepts in the "Common Object Request Broker Architecture: Core Specification". It is by no means an introduction to CORBA, but contains some important information that helps understand how CORBA interception service works and compare the CORBA interception service and the RMI interception service introduced in this work.

CORBA is an open standard for distributed object development defined by the Object Management Group (OMG). *"CORBA manages details of component interoperability, and allows components to communicate with one another despite different locations, platforms and implementing languages"* [16, 31]. The interface, which is defined by IDL (CORBA Interface Definition Language), is the only way that components communicate with each other.

*"The most important part in CORBA architecture is the Object Request Broker (ORB). The ORB is the middleware that establishes the client-server relationships between components. Using an ORB, a client can request services from a server object, whose location and implementation are completely transparent"* [16]. *"The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect that is not reflected in the object's interface"* [16]. In this way, the ORB provides interoperability among applications on distributed machines in heterogeneous environments and seamlessly interconnects multiple components [16, 31]. Figure 6.1 shows the components of ORB architecture in CORBA applications:

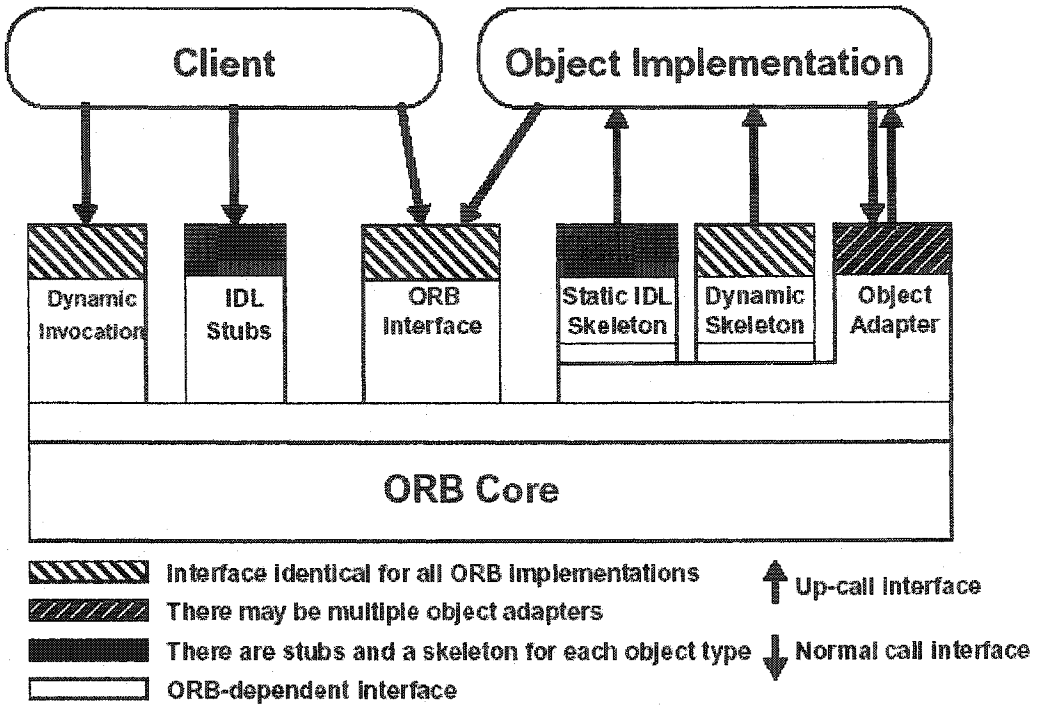


Figure 6.1: Main components of the ORB architecture and their interconnections [16]

In this architecture, a Client can send a request to the server object either by using the Dynamic Invocation interface or an IDL stub. The Client can also directly communicate with the ORB interface for some services. The Object Implementation receives a request as an up-call either through the IDL generated skeleton or through a dynamic skeleton.

The Object Implementation may call the Object Adapter and the ORB for services. The client performs a request by having access to an Object Reference to an object implementation, initiates the request by calling IDL stubs or by constructing the request dynamically. The receiver of the message cannot tell how the request is invoked because the dynamic and stub interface for invoking a request have the same signature. The ORB intercepts the request, locates the appropriate implementation, transmits parameters, and passes control to the Object Implementation through an IDL skeleton or a dynamic skeleton. While performing the request, the object implementation may obtain some services from the ORB through the Object Adapter. When the request is complete, control and output values are returned to the client [16].

### 6.1.1 OMG IDL

*"The OMG Interface Definition Language (IDL) is the language used to describe the interfaces that client objects call and object implementations provide. An interface definition written in OMG IDL completely defines the interface and fully specifies each operation's parameters. An OMG IDL interface provides the information needed to develop clients that use the interface's operations"* [16].

### 6.1.2 Java IDL

Java IDL is the binding of the OMG IDL concepts to Java programming language. *"Java IDL adds CORBA (Common Object Request Broker Architecture) capability to the Java platform, providing standards-based interoperability and connectivity. Java IDL enables distributed Web-enabled Java applications to transparently invoke operations on remote network services using the industry standard OMG IDL (Object Management Group Interface Definition Language) and IIOP (Internet Inter-ORB Protocol) defined by the Object Management Group. Runtime components include an Object Request Broker (ORB) for distributed computing using IIOP communication"* [32].

Detailed explanations of the Dynamic Invocation, IDL Stubs, ORB interface, Static IDL Skeleton, Dynamic Skeleton and Object Adapter are beyond the scope of this research; interested users could refer to [16] for more information.

## 6.2 CORBA Portable Interceptors

As introduced at the beginning of this thesis, our testing components can be reused in CORBA applications with only minor changes. The recent CORBA specification supports *portable interceptors*, through which one can easily write and attach portable ORB hooks that will intercept any ORB-mediated invocation. The following part is not intended to present an overview of CORBA *portable interceptors*; it rather focuses on some of their features, which are necessary to understand how our testing components can be reused in a CORBA environment.

The CORBA Object Request Broker (ORB) provides hooks -- *Portable Interceptors*, through which ORB services can intercept the normal flow of execution of the ORB. These portable interceptors provide a mechanism for plugging in additional ORB behavior, or, by modifying the communications between client and server, for modifying the behavior of the ORB [18]. CORBA currently defines three types of interceptors, i.e., IORInterceptor, ClientRequestInterceptor, and ServerRequestInterceptor [16, 18]. In the testing architecture that we will implement in CORBA, we will use the latter two, which are called request interceptors in general.

"*A request Interceptor is designed to intercept the flow of a request/reply sequence through the ORB at specific points so that services can query the request information and manipulate the service contexts which are propagated between clients and servers*" [16, 18]. Figure 6.2 illustrates the simplified ORB architecture with Portable Interceptors:

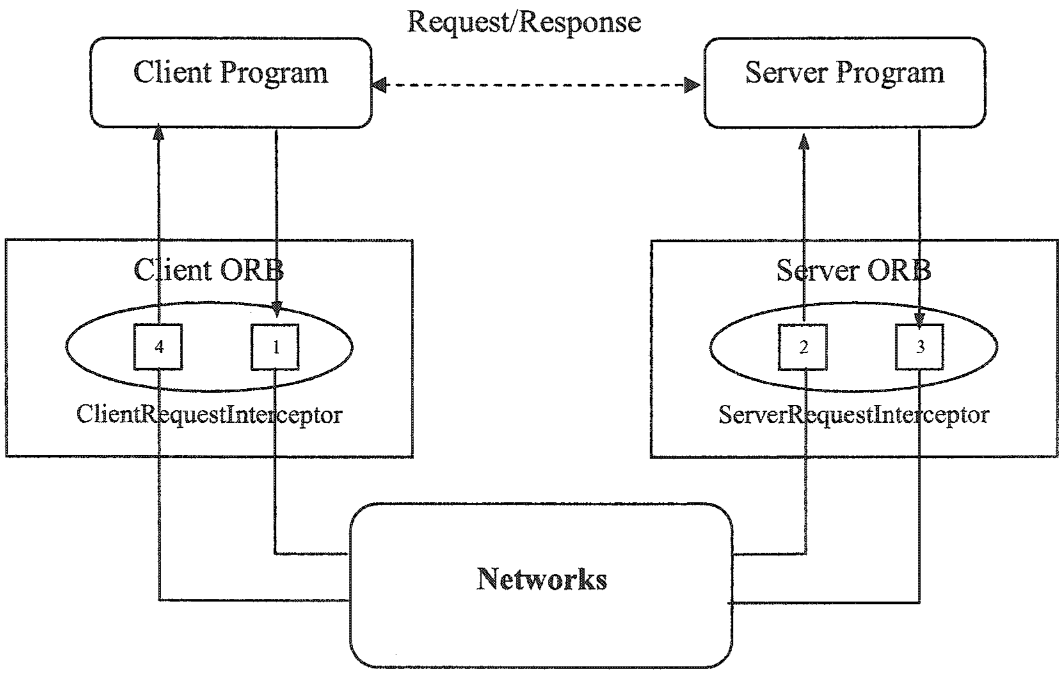


Figure 6.2: Simplified ORB architecture with Portable Interceptors

A ClientRequestInterceptor intercepts the flow of a request/reply sequence through the ORB on the client side, while a ServerRequestInterceptor intercepts the flow of a request/reply sequence through the ORB on the server side. During a request/reply lifecycle, each request Interceptor is called at a number of interception points, among which we are only interested in the *send_request* and *receive_reply* in the ClientRequestInterceptor, and the *receive_request* and *send_reply* in the ServerRequestInterceptor. The flow of control for exercising these interception points is very similar to those we have described in Java RMI interception services. In fact, the interception services we build in Java RMI follow the working mechanism of CORBA *portable interceptors*.

A user-defined interceptor must implement, either directly or indirectly the ClientRequestInterceptor and/or ServerRequestInterceptor, in order to be a means by which ORB services gain access to ORB processing and be effectively becoming part of the ORB. Request interceptors must be registered with an associated ORBInitializer object, which implements the ORBInitializer interface. When an ORB is being initialized, it shall call each registered ORBInitializer, passing it an ORBInitInfo object, which is used to register its interceptor(s).

# 7. Reusing the Testing Components in CORBA

In section 2.2, we have mentioned that we provided an interception service, which allows application testers to plug in their testing code into the RMI middleware layer, by implementing and extending a set of CORBA-like classes, and starting this interception service in a way very similar to that of CORBA. In the following, we summarize the similarities between CORBA interception service and the interception service we have provided in Java RMI:

| | **CORBA Interception Service** | **Java RMI Interception Service** |
|---|---|---|
| **Implementing User-defined Interceptors** | By implementing the ClientRequestInterceptor and/or ServerRequestInterceptor interfaces | By extending the RMIClientInterceptor and/or RMIServerInterceptor classes |
| **Interception Points and Flow of Control** | 1. send_request,<br>2. receive_request,<br>3. send_reply,<br>4. receive_reply | 1. send_request,<br>2. receive_request,<br>3. send_reply,<br>4. receive_reply |
| **Accessing Request/Reply Information** | By calling methods on the ClientRequestInfo/ ServerRequestInfo interfaces | By calling methods on the ClientRequestInfo/ ServerRequestInfo interfaces |
| **The Way to Register Interceptors** | By implementing the ORBInitializer interface and add interceptors using an ORBInitInfo object | By implementing the InterceptorInitializer interface and add interceptors using an RMIInterceptorServer object |
| **Multiple Interceptors** | Support | Support |
| **Start the Interception Service** | By specifying the System property: *org.omg.PortableInterceptor.ORBInitializerClass* in the command line when running a Java program | By starting the interceptor server with the argument: *ca.uwindsor.kunwang.rmi.interceptor.InterceptorInitializerClass* in the command line |

Table 7.1: Comparisons between CORBA and Java RMI for implementing and registering interception services

When implementing user-defined interceptors in Java RMI, we just use different class names from those in CORBA for client and server interceptors. The reason that we define these "*root*" interceptors as classes rather than interfaces is that we must include the instance variable *pid* (process id) in user-defined interceptors for testing purposes. These two "*root*" interceptors define a unique constructor that takes a *pid* as parameter, and this forces users to initialize the process id when an interceptor is created. In this way, we can assure the process id is an integral part of the interceptors. The interception points and the flow of control of a request/response are exactly the same as those in CORBA. When accessing request or reply information in Java RMI, we also adopt the same names for the *RequestInfo* objects as those in CORBA. When registering user-defined interceptors in Java RMI, we choose different names for the interceptor initializer interface and the object used for adding interceptors. In the ORBInitializer interface, users must implement the *pre_init* and/or the *post_init* methods, while in the InterceptorInitializer, users only need to implement the *init* method. For the ORBInitInfo and the RMIInterceptorServer objects, the methods for adding client or server interceptors are different only in names. Finally, there are certain differences between the Java RMI and CORBA when starting the interception service. In Java IDL, there is a pre-defined system property: *org.omg.PortableInterceptor.ORBInitializerClass* for specifying the fully qualified class name of the user-implemented *ORBInitializer*. However, we do not have such a pre-defined system property, so we provide a similar property: *ca.uwindsor.kunwang.rmi.interceptor.InterceptorInitializerClass,* which is specified in the batch file for starting the Interceptor Server. The user implemented fully qualified name of the InterceptorInitializer must be specified in the configuration.xml file.

Now, with the help of CORBA portable interceptors, one can easily incorporate our testing components into the CORBA architecture. When adopting CORBA architecture, we consider using the Java ORB as our underlying CORBA ORB implementation. The reason that we have chosen the Java ORB at this time is that it is free software shipped with J2SE 1.4.1. The Java ORB in the J2SE 1.4.1 platform complies with the CORBA 2.3.1 specification and supports the IDL to Java language mapping specification, the

Interoperable Naming Service specification and the Portable Interceptor specification. Apparently, CORBA-based applications written only in Java language are also platform-independent because of the portable feature of Java language and also because the Java ORB is shipped with J2SE 1.4.1 and can be installed on almost any operating system. The Sun Java ORB is a good and free ORB implementation; however, there are many advanced features of CORBA that are missing. For example, there are no Transaction Service or Event Service in Sun Java ORB, neither does it support IDL to C++ language mapping, i.e., it cannot translate IDL to C++ code, which means that CORBA applications using Java ORB as the middleware cannot  incorporate systems written in other languages.

Of course, we may use different ORB implementations as long as they support CORBA Core Specifications such as Naming Service, Portable Interceptor and the IDL to Java language mapping, etc. Many good ORB products are available in the market, in which VisiBroker from Inprise Corp., Orbix from IONA Technologies and ORBacus from Object-Oriented Concepts, Inc. are leading ones. For example, ORBacus is a fully CORBA-compliant ORB that is distributed as source code and is free for non-commercial use. It supports more CORBA specifications than Java ORB, such as Event Service and IDL to C++ translation. ORBacus also has different versions for different platforms, so it provides users the ability to develop real distributed and heterogeneous applications.

However, no matter which ORB is chosen, Java IDL (which is a technology for CORBA programmers who want to program in the Java programming language based on interfaces defined in CORBA Interface Definition Language) is structured with a "pluggable ORB" architecture, which allows us to instantiate ORBs from other vendors from within the Java Virtual Machine. This is a very nice property of Java IDL; it means that a CORBA application written in Java only has to change very few pieces of code (or none at all) in order to be moved from one ORB to another ORB implementation. This is accomplished through setting environment variables, or system properties, or at run time through the use of a Properties or String[] object.

Now, we discuss how to move the testing components from Java RMI to CORBA. To insert interceptors into ORBs, we simply make the interceptor in each ORB implement both the ClientRequestInterceptor and the ServerRequestInterceptor (these two classes are available in Java API), since processes will communicate in a peer-to-peer manner and we need to intercept both incoming and outgoing calls at the same time. In order to reuse the testing components in the new environment, the only major job we need to do is to modify the Java RMI remote objects to CORBA objects, and change the way that objects are registered and located. Since the control logic of those remote objects and non-remote objects are the same as that of Java RMI, most part of these components can be reused without any change. This can be achieved by defining IDLs, generating the stubs and skeletons and making the testing component implementation extend those skeletons. We also need to do some extra work to deal with the difficulties caused by CORBA's inability to support most data types in Java API. For example, to map a Java Vector in CORBA, we have to define a new data type using struct (similar to that in C language) and sequence. These mappings and changes can be done in several ways. One possible solution is to modify the object implementations directly and recompile. With these modifications, we can easily plug in the test control mechanism into the CORBA ORB and move the testing components to a CORBA environment.

| | **Things to be Changed in Java RMI Interception Service** | **Resulting Changes in CORBA Interception Service** |
|---|---|---|
| **Remote Service Definition** | Writing a Remote Interface definition for each type of remote objects | Defining all services of different types of remote objects in a single CORBA IDL file |
| **Registering and Locating Remote Objects** | Using the Java Naming Service: starting a built-in Java naming server (RMIRegistry), and use API in the Java RMI packages | Using the CORBA Naming Service: starting a built-in CORBA server (ORBD), which provides bootstrap services, and use API in the Java IDL packages |
| **Registering Objects on other Machine** | Does not allow remote objects to be registered on a machine(s) other than the one that their implementations reside | Allow remote objects to be registered on a machine(s) other than the one that their implementations reside |
| **Data Type Mappings** | Vector, Hashtable, and File... | sequence, struct sequence, and array of strings |
| **Method Overloading** | Allowed in Java language | Change the same method names into different names in IDL |
| **Obtaining Information (e.g pid, iname, mname) For Testing** | Can get iname and mname by calling methods on the ClientRequestInfo/ ServerRequestInfo interfaces; pid is obtained as an inherited instance variable from the RMIClientInterceptor and/or RMIServerInterceptor | Can obtain pid by calling *getProcessId()* on *TestController;* can obtain *mname* by calling *getMname()* on the *RequestInfo* object; cannot get *iname* |
| **Heterogeneity in Running Environment** | Does not support a heterogeneous running environment. Applications Under Test and Test Components must be implemented in Java Language | Support a heterogeneous running environment. Both Applications Under Test and Test Components can be implemented in different programming languages |

Table 7.2: Major Changes of Testing Components from Java RMI to CORBA

In table 7.2, we only listed some major changes in order to migrate these testing components into a CORBA environment. Specifically, one of the major problems in moving the testing environment from Java RMI to CORBA is to obtain the *iname* (invoked remote interface name). In Java RMI, we can easily get the *iname and mname* (invoked method name) by calling the methods defined in the *RequestInfo* interface. We can also obtain the *pid* (process id) by using the initialized instance variable *pid*. In CORBA, the only way to get access to the request/response information is through the *ClientRequestInfo* and *ServerRequestInfo* interfaces, but unfortunately, these interfaces do not provide a way to obtain the invoked remote interface name. So, in the case that there are more than one remote interface that might be invoked in an AUT, we have to use "hard code" method in the portable interceptors to obtain the remote interface name, which will be used as a parameter in the call to the *permissionRequest* of the *TestController*. We can obtain this interface name by comparing the invoked method name; however, like what we have done for the *RequestInfo* in Java RMI, this method is also based on the assumption that there cannot be identical public method definitions in those implemented interfaces.

## 7.1 ClientRequestInfo/ServerRequestInfo in Java RMI and CORBA

Both in Java RMI and in CORBA, each interception point is given an object through which the Interceptor can access request information. Client-side and server-side interception points are concerned with different information, so there are two information objects: *ClientRequestInfo* is passed to the client-side interception points and *ServerRequestInfo* is passed to the server-side interception points. But there is information that is common to both, so they both inherit from a common interface: *RequestInfo*. In this section, we will compare the information that can be obtained via *RequestInfo* objects in Java RMI with the information that can be obtained via *RequestInfo* objects in CORBA. One thing should be noted here is that we do not intend to compare all the information that can be obtained from these objects, and we only list those properties that are relevant to software testing.

| **Environment** / **Available Information** | **Java RMI** | **CORBA** | |
|---|---|---|---|
| | | **Core Specification** | **Java IDL** |
| ***target*** | yes | yes | yes |
| ***iname*** | yes | no | no |
| ***operation*** | yes | yes | no |
| ***params*** | yes | yes | no |
| ***result*** | yes | yes | no |
| ***clientHost*** | yes | no | no |
| ***contexts*** | no | yes | no |

Table 7.3: Available Information from the *RequestInfo* objects in different environments

In table 7.3, we list some useful properties that can be obtained from the *RequestInfo* objects in different computing environments. The column "CORBA" is divided into two groups: core specification and Java IDL. This is because the OMG IDL has different mappings for different implementation languages, and the Java IDL is the binding of the OMG IDL concepts to Java programming language, and thus a Java implementation of the CORBA core specification. The attributes (such as *target, operation,* and *contexts,* etc.) which are defined in the CORBA core specification are not accessible in the current Java environment. In this environment, when these attributes are accessed, NO_RESOURCES exception will be raised with a standard minor code of 1 [16]. Now, we explain this available information in detail.

*Target* represents the server object which the client called to perform an operation. In Java RMI, this *target* is a remote server object that implements the interface *java.rmi.Remote*, while in CORBA, this *target* is an implementation of type *org.omg.CORBA.Object.* The *iname* is only defined in the interception service in Java RMI, which represents the remote interface name that is being invoked by the client. The *operation, params,* and *result* can be obtained in both Java RMI and CORBA infrastructure, which respectively represent the operation (method) that is being invoked, the parameters that are passed into this operation and the result of this operation invocation. The operation in both Java RMI and CORBA are method names of type *String*. The *params* in Java RMI is an array of Java Objects, while the *params* in CORBA is a *ParameterList* object, containing the arguments on the operation being invoked. The *result* is a *java.lang.Object* in Java, but is an object of type *org.omg.CORBA.Any* in CORBA. The *clientHost* can only be obtained from the interception service of Java RMI and denotes the IP address of the client host making requests. The *contexts* is only available in CORBA and is a *ContextList* object describing the contexts that may be passed on this *operation* invocation.

Here, we did not talk about the object id, a crucial property that is useful not only for software testing, but also for dynamic monitoring and analysis of an object-oriented distributed program. For example, when there are multiple objects implementing the same interface in our test scenarios, we can specify different objects by using object ids in the test scenario document, and further determine requests/responses from individual objects on the fly. The object id can be obtained in both computing environments. In Java RMI, the object id is represented in the form of a *java.rmi.server.ObjID*, while in CORBA, it is an array of bytes describing the target of the operation invocation. But unfortunately, neither of these object ids are human readable, and thus cannot be used in the test scenario specification.

## 7.2 Comparisons of the Design Principles between Java RMI Interception Services and CORBA Portable Interceptors

| | RMI Interception Services | | | CORBA Portable Interceptors |
|---|---|---|---|---|
| | Dynamic Proxy * | SSL | RRL | |
| **Redirect a call** | Yes | No | Yes | Yes |
| **Alter arguments** | Yes | Yes | Yes | No |
| **Make object invocations** | Yes | Yes | Yes | Yes |
| **Delay a request/reply** | Yes | Yes | Yes | Yes |
| **Generate own reply** | Yes | Yes | Yes | No |
| **Piggyback Additional Info.** | No | No | No | Yes |

Table 7.4: Comparisons of the design principles between Java RMI Interception Services and CORBA Portable Interceptors

* RMI Interception Services using dynamic proxy techniques can only be used at client side and only for those remote objects that are looked up through a Naming Service.

The above table compares some design principles of Java RMI interception services with those of the CORBA Portable Interceptors. The CORBA Portable Interceptor architecture is designed to:

- Redirect a request to another target by raising a ForwardRequest exception
- Affect the outcome of a request by raising a system exception or redirect a reply to another target by raising a ForwardRequest exception
- Make object invocations itself before allowing the current request to execute, and thus can be used to delay a request or a reply.
- Piggyback Service-specific information to be passed implicitly with requests and replies

In some circumstances, Portable Interceptors are not sufficient to meet some specific requirements in different applications. In particular, the limitations of Portable Interceptors can be summarized as follows [34]:

- Cannot generate own replies to intercepted requests.
- Cannot affect a request by changing a parameter specified by the client.
- Can redirect a request or a reply only by raising an exception.

We will explain how the interception services in Java RMI can overcome the above three limitations, and of course, the Java RMI interception service also has its own limitations. In the following, we will discuss these issues in three different layers of the Java Run-time system, into which an interception service may be injected.

### 7.2.1 Interception using Dynamic Proxy technique

As we have discussed in section 4.1, the Dynamic Proxy technique can be viewed as a hook-up mechanism and can be used as a type of interception service together with some additional interceptor interfaces (such as *RMIClientInterceptors* and *RMIServerInterceptors*). However, the limitation of this technique is that interception services implemented using dynamic proxy technique can only be used in the client side and only for looked-up objects. As a result, interception service using dynamic proxy technique cannot be used to transmit additional information from client side to server side (because there no corresponding proxy objects on the server side).

By using dynamic proxy, a remote method invocation will be encoded and dispatched to the *java.lang.reflect.InvocationHandler*, and further directed to the interception points. Thus, redirecting a request, modifying arguments and generating a response can be realized relatively easily by using Java Reflection in these interception points. In the implementation of an interceptor, any kind of object invocations can be made and thus a request/reply can be delayed or blocked for arbitrary time (e.g., by calling *Thread.sleep*).

### 7.2.2 Interception in SSL (Stub and Skeleton Layer)

In this layer, we can build the interception services into the stubs and skeletons. At the point where a request or reply is intercepted, the control flow has actually entered the stub or skeleton of the target object and the method has been invoked. So, interception service in this layer cannot be used to redirect this call to another object. The classes that allow users to plug in their own code in SSL are *RMIClientInterceptors* and *RMIServerInterceptors*. The arguments, target object and return values of a request can be accessed (both read and write) by using the *RequestInfo* object. Thus, we may alter the arguments or return value by modifying the way that stubs and skeletons are generated. At any interception point in the SSL layer, any kind of object invocations can be made and thus a request/reply can be delayed or blocked for arbitrary time.

### 7.3.3 Interception in RRL (Remote Reference Layer)

The interfaces that allow users to alter request information in RRL are exactly the same as those of SSL, except that we modified the UnicastRef and UnicastServerRef classes to introduce an interception mechanism. The parameters and return value can be altered by using Java Reflection. At any interception point in the RRL layer, any kind of object invocations can be made and thus a request/reply can be delayed or blocked for arbitrary time.

# 8. Empirical Evaluation

In this chapter, we analyze some experimental results to evaluate the functionality and performance of our reproducible testing environment. In section 1, we run this testing environment using the previous Online Conference example as the application under test. We demonstrate how this testing environment forces the AUT to execute desired paths by comparing the experimental results before and after adopting our control algorithm. In section 2, we run this Online Conference example on both the distributed testing environment and the centralized testing environment, which have the same functionality but with different system infrastructures. This experiment is to compare the performance of these two architectures and to show users how to choose one architecture instead of the other in different situations.

As introduced in Section 3.2, the AUT, Online Conference example that we will use in the following experiments is an implementation of the distributed bakery algorithm, and involves two processes (namely Peer-0 and Peer-1) competing to talk (entering its critical section). In both experiments, these two processes together with their own local test components run on two separate machines and communicate with each other in a peer to peer manner. Both machines have the same operating systems (Microsoft Windows 2000, Profession Edition) and the Java Platforms (Java Development Kit 1.4.1_01). The central test components, i.e. the Central Test Oracle and the Central Communicator are two separate processes that can be deployed on any other machine(s). But in our experiments, these two components are running on the same host as Peer-0 because Peer-0's host has a better hardware configuration (larger memory).

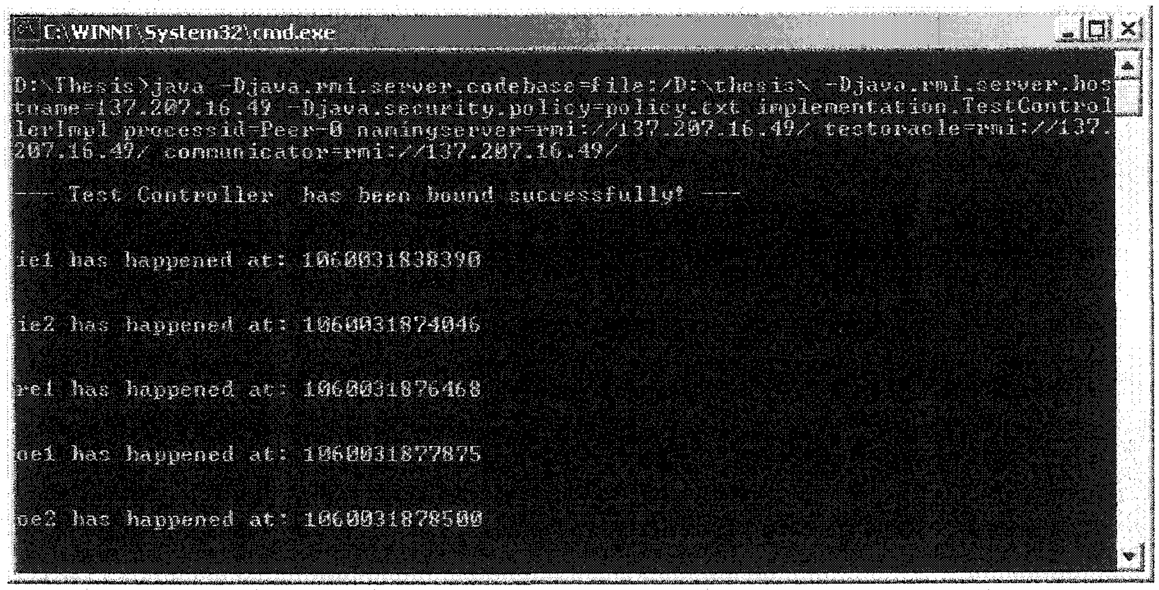
## 8.1 Running Online Conference example

### 8.1.1 Running the Online Conference without Adopting the Control Algorithm

In this experiment, we run the Online Conference example based on the testing environment described in Chapter 5. However, we do not use any control algorithm in the test controller. That is, whenever the test controller receives a request from an input event or remote event (through a test driver or an interceptor), it only analyzes the request and records the time at which this event occurs, without checking the test constraint for permission. More precisely, the testing environment we described in Chapter 5 is only used as a software instrumentation framework, which dynamically monitors and logs the program execution without touching any implementation of this program.

Without artificial control, Peer-0 and Peer-1 run at their own speeds independently, and the execution paths can be arbitrary. Figure 8.1 and figure 8.2 show the snap shots of running Peer-0 and Peer-1 without adopting our control algorithm.

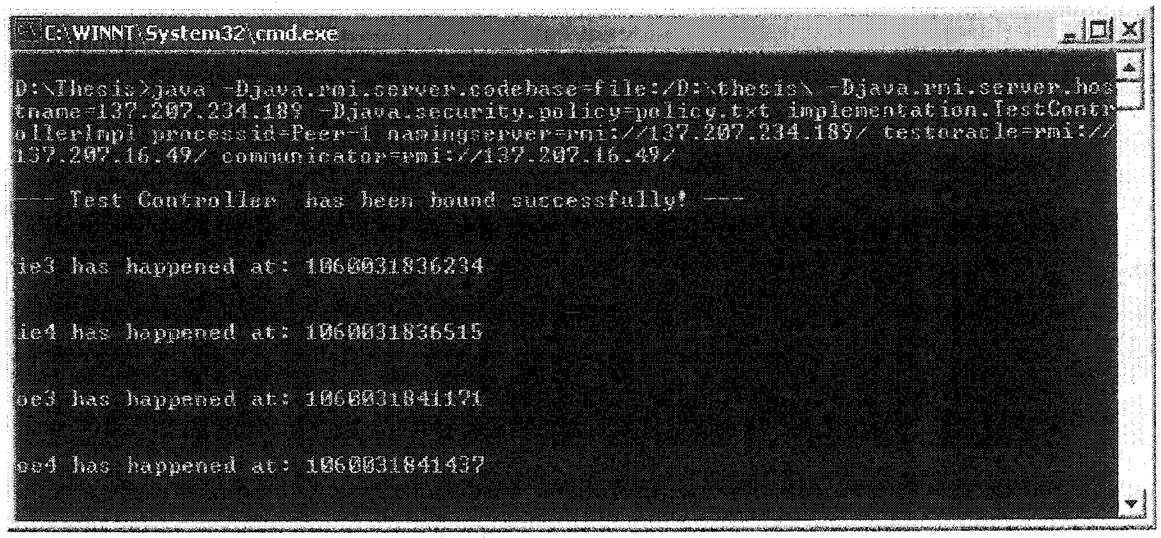
Peer-0:



```
C:\WINNT\System32\cmd.exe

D:\Thesis>java -Djava.rmi.server.codebase=file:/D:\thesis\ -Djava.rmi.server.hos
tname=137.207.16.49 -Djava.security.policy=policy.txt implementation.TestControl
lerImpl processid=Peer-0 namingserver=rmi://137.207.16.49/ testoracle=rmi://137.
207.16.49/ communicator=rmi://137.207.16.49/

--- Test Controller  has been bound successfully! ---

ie1 has happened at: 1060031838390

ie2 has happened at: 1060031874046

re1 has happened at: 1060031876468

oe1 has happened at: 1060031877875

oe2 has happened at: 1060031878500
```

Figure 8.1: Result of Running Peer-0 without Adopting Control Algorithm

Peer-1:



```
D:\Thesis>java -Djava.rmi.server.codebase=file:/D:\thesis\ -Djava.rmi.server.hos
tname=137.207.234.189 -Djava.security.policy=policy.txt implementation.TestContr
ollerImpl processid=Peer-1 namingserver=rmi://137.207.234.189/ testoracle=rmi://
137.207.16.49/ communicator=rmi://137.207.16.49/

--- Test Controller  has been bound successfully! ---

ie3 has happened at: 1060031836234

ie4 has happened at: 1060031836515

oe3 has happened at: 1060031841171

oe4 has happened at: 1060031841437
```

Figure 8.2: Result of Running Peer-1 without Adopting Control Algorithm

The last columns of these two snap shots show the time that a specific event happens, represented by the number of milliseconds since January 1, 1970, 00:00:00 GMT. One thing that should be taken into account is that the time for each event is the local CPU time of each host. We compare the orders of events based on the assumption that both CPUs' time is exactly the same. By comparing the time at which each event occurs, we can draw the event sequence diagram for these two running processes:

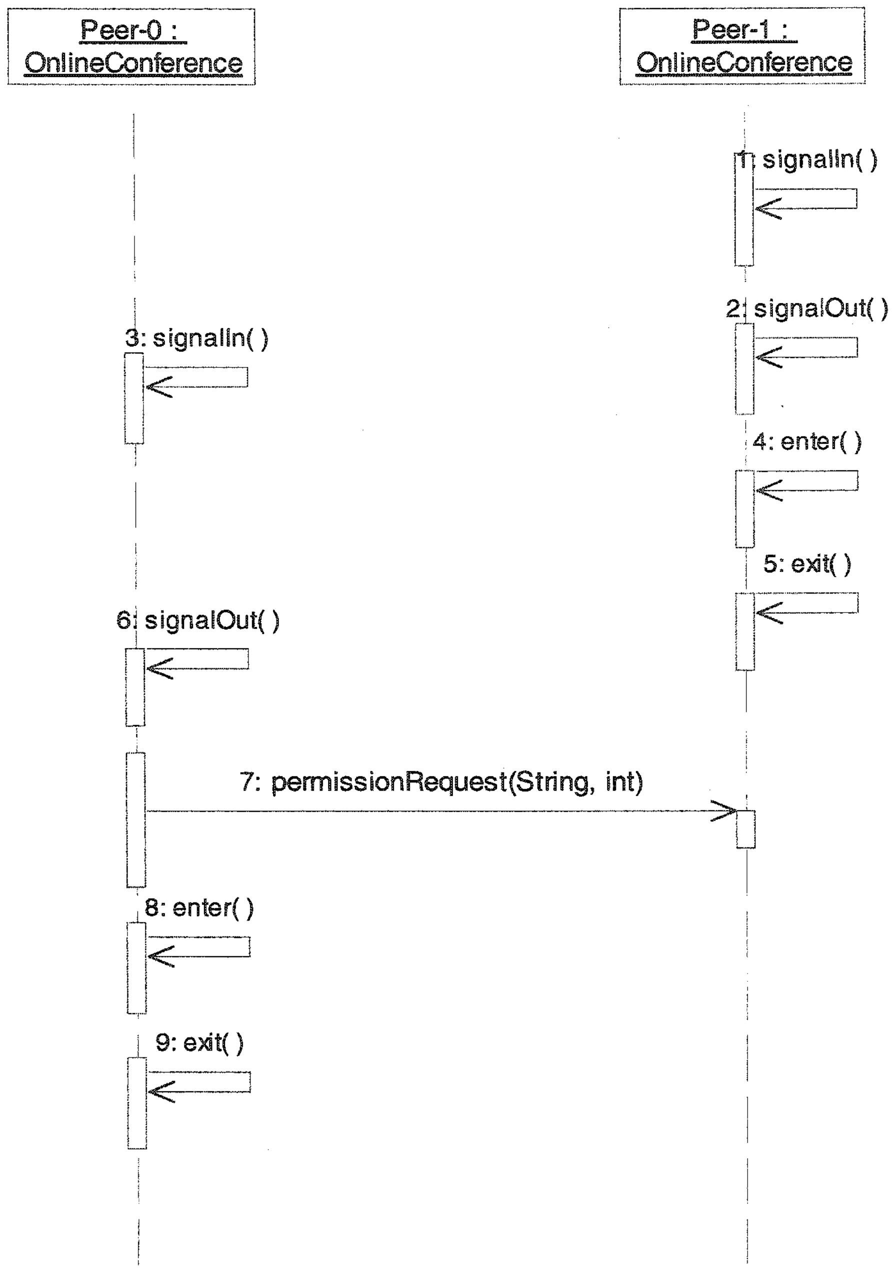


Figure 8.3: Event Sequence when running Peer-0 and Peer-1 without Control

In the above figure, the numbers on the event names denote the orders of occurrences of the events. The event names: *signalIn, singalOut*, *permissionRequest*, *enter*, and *exit* correspond to the following kinds of events:

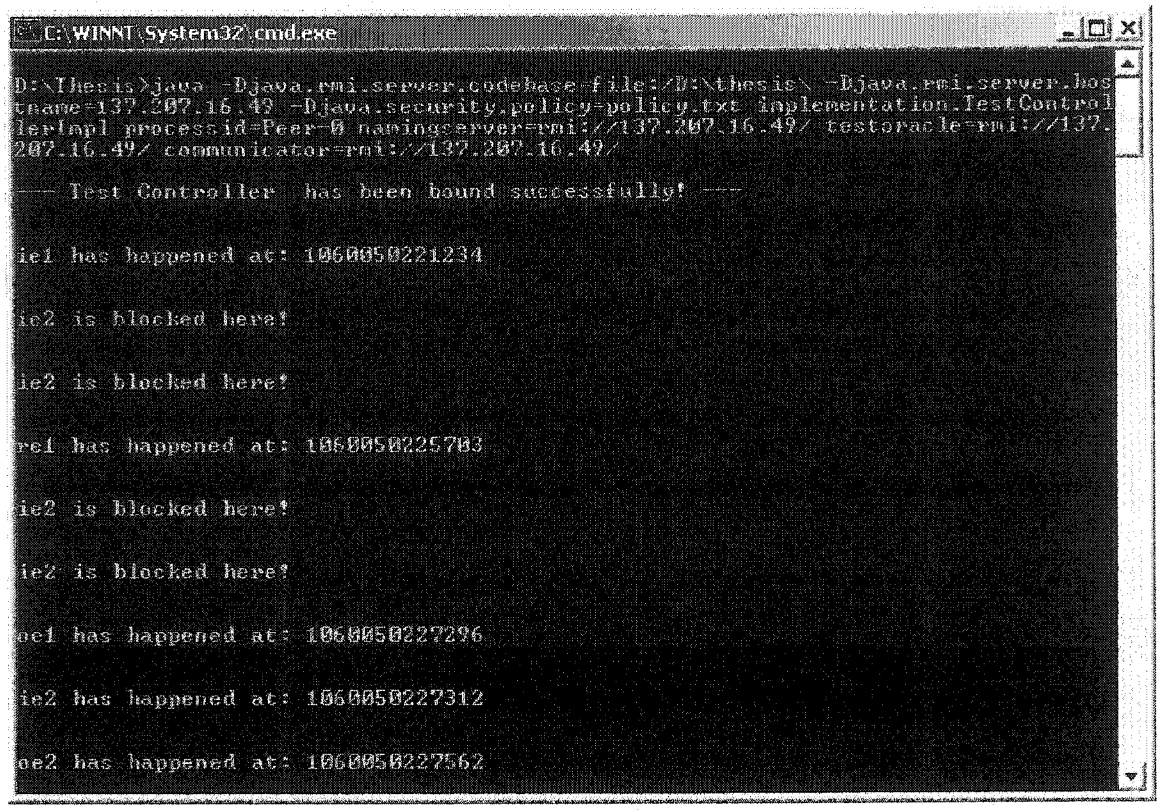
- input events signaling the willing to speak
- input events signaling the willing to finish speaking
- remote events requesting permission from other processes
- output events denoting the start of speaking and
- output events denoting the end of speaking.

According to the test scenario we defined in section 3.2, the nine numbered events in the above figure respectively represent *ie3, ie4, ie1, oe3, oe4, ie2, re1, oe1,* and *oe2*. From this event sequence diagram, we can easily see that the order of events does not satisfy the test constraint we specified in figure 3.2 (e.g., *ie1* should happen before *ie3*). If we run this program several times, it may display different event sequences, but we cannot guarantee that these event sequences satisfy the test constraint. This nondeterminism is a typical characteristic of a concurrent distributed program.

### 8.1.2 Running the Online Conference by Applying the Control Algorithm

In the following, we will demonstrate how these two processes are forced to execute according to the desired paths by applying the control algorithm introduced in 5.3. Figure 8.4 and figure 8.5 show the snap shots of running Peer-0 and Peer-1 with the control algorithm.

Peer-0:

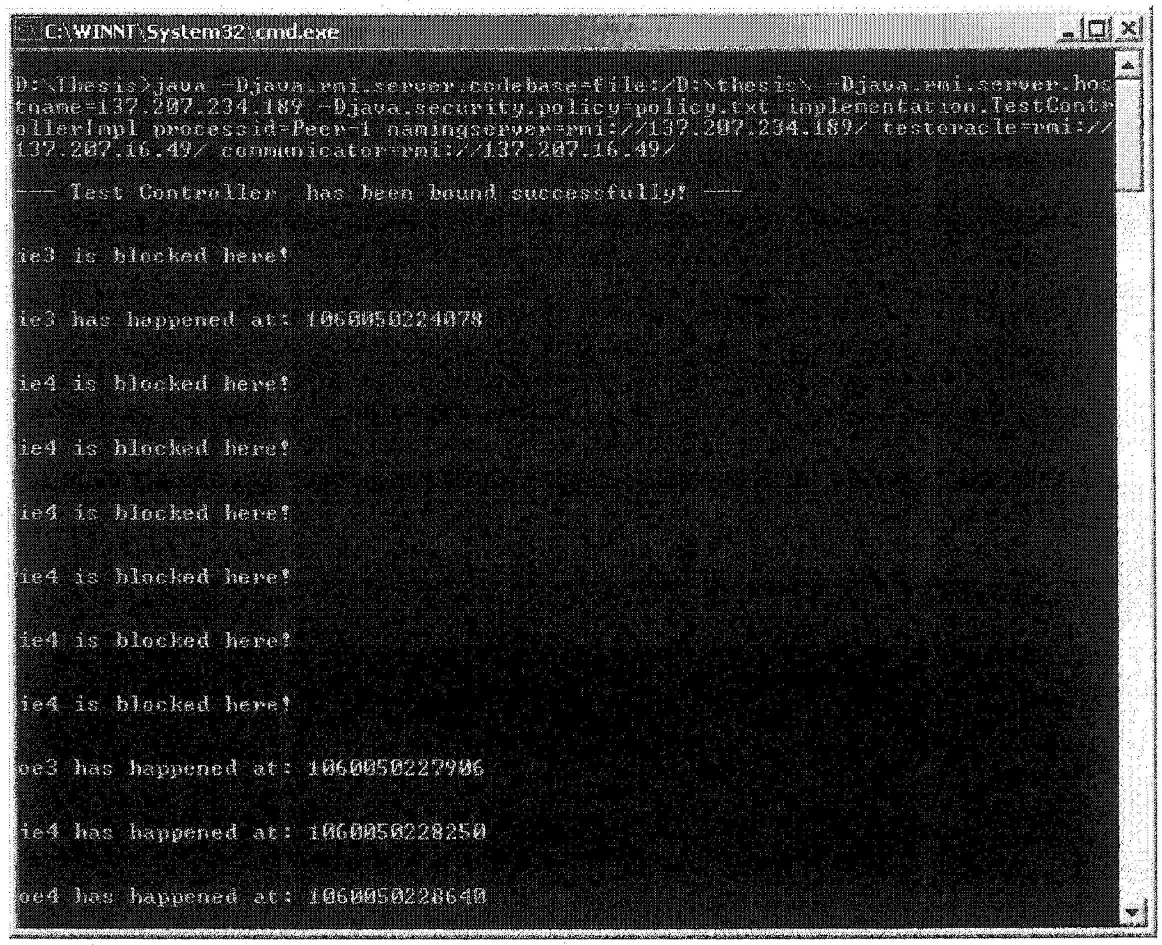


C:\WINNT\System32\cmd.exe

```
D:\Thesis>java -Djava.rmi.server.codebase=file:/D:\thesis\ -Djava.rmi.server.hos
tname=137.207.16.49 -Djava.security.policy=policy.txt implementation.TestControl
lerImpl processid=Peer-0 namingserver=rmi://137.207.16.49/ testoracle=rmi://137.
207.16.49/ communicator=rmi://137.207.16.49/

--- Test Controller  has been bound successfully! ---

ie1 has happened at: 1060050221234

ie2 is blocked here!

ie2 is blocked here!

re1 has happened at: 1060050225703

ie2 is blocked here!

ie2 is blocked here!

oe1 has happened at: 1060050227296

ie2 has happened at: 1060050227312

oe2 has happened at: 1060050227562
```

Figure 8.4: Result of Running Peer-0 with Control Algorithm

Peer-1:



```
D:\Thesis>java -Djava.rmi.server.codebase=file:/D:\thesis\ -Djava.rmi.server.hos
tname=137.207.234.189 -Djava.security.policy=policy.txt implementation.TestContr
ollerImpl processid=Peer-1 namingserver=rmi://137.207.234.189/ testoracle=rmi://
137.207.16.49/ communicator=rmi://137.207.16.49/

--- Test Controller  has been bound successfully! ---

ie3 is blocked here!

ie3 has happened at: 1060050224078

ie4 is blocked here!

ie4 is blocked here!

ie4 is blocked here!

ie4 is blocked here!

ie4 is blocked here!

ie4 is blocked here!

oe3 has happened at: 1060050227906

ie4 has happened at: 1060050228250

oe4 has happened at: 1060050228640
```

Figure 8.5: Result of Running Peer-1 with Control Algorithm

From figure 8.4 and figure 8.5, we observe that certain events are blocked before they are allowed to happen. This is because the events we are interested in controling must happen according to the orders we specified in the test constraint, which is read into the test controller from the central test oracle. On one hand, all involved test controllers will update current state of this running program whenever a certain event specified in the test constraint happens. On the other hand, a certain event keeps trying to check if it is allowed to happen each time after the program state is updated. If it is not granted the permission, this event will be blocked again until it is allowed to happen. Again, by comparing the time of occurrence of each event, we can draw the event sequence diagram for these two running processes with the control algorithm:

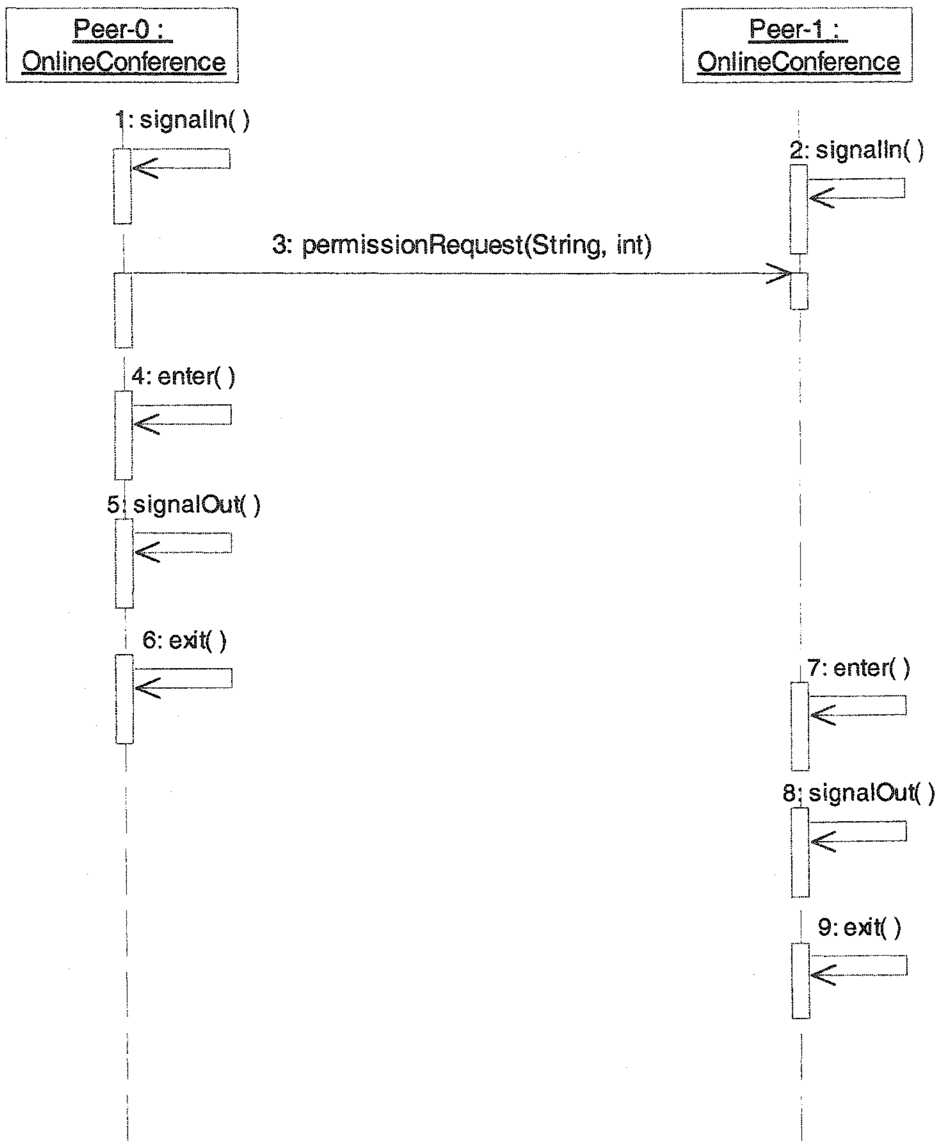


Figure 8.6: Event Sequence when running Peer-0 and Peer-1 with the Control Algorithm

Similar to the previous event sequence diagram, the nine numbered events in figure 8.6 respectively represent *ie1, ie3, re1, oe1, ie2, oe2, oe3, ie4,* and *oe4*. These nine events happen exactly as this sequence. Apparently, this specific event sequence satisfies the event order we have specified in the test constraint (i.e., *ie1* happen before *ie3* and *ie3* happen before *re1*).

### 8.1.3 Comparing time cost when running a test using control

In this experiment, we evaluate the time cost caused by applying the control mechanism to the testing architecture. Again, *without control* actually means that we do not use the control algorithm introduced in section 5.3, but we still use the testing components for software instrumentation -- monitoring and logging purposes. In such a case, there is additional time cost caused by the software instrumentation. Since this time cost is negligible compared with that of the control algorithm, we simply omit it. The following table contains time cost (in millisecond) that we choose from independently running the Online Conference sample application by each approach three times. The experiment settings are exactly the same as those we described in the introduction of this chapter. In this experiment under these settings, the AUT (Online Conference) consists of two processes and the test scenario contains nine events and one constraint, which have been defined in Section 3.2. For each run, we calculate the time spent from starting the whole application (after both processes on both machines start) to the receipt of the final output (*succeeds* or *fails*) at the central test oracle. From these data, we observe that running a program with adopting control always has some additional time cost, and the average cost for the Online Conference example is 1313 milliseconds.

| | **With Control** | **Without Control** |
|---|---|---|
| **First Run** | 4297 ms | 1750 ms |
| **Second Run** | 3172 ms | 2969 ms |
| **Third Run** | 4360 ms | 3172 ms |

Table 8.1: Time cost by adopting control algorithm

## 8.2 Performance Evaluation

In this section, we compare the performance of a distributed testing architecture with that of a centralized one. We will also discuss how to choose one architecture instead of the other in different situations. To compare the performance of both architectures, we need to modify the existing distributed testing architecture into a centralized one, i.e., using a central test controller instead of a local controller. Because there is only one central test controller, which contains both the test constraint and the current running state of the program, we do not need the communicator to broadcast updates of states. This modification involves three major changes to the implementation of the existing testing environment: 1) the way that test drivers and interceptors look up the test controller, 2) the way that the central controller updates the program running state and 3) the way that test drivers obtain information such as the number of processes under test. We will not further discuss the details of these changes. Figure 8.7 shows the resulting centralized testing environment:

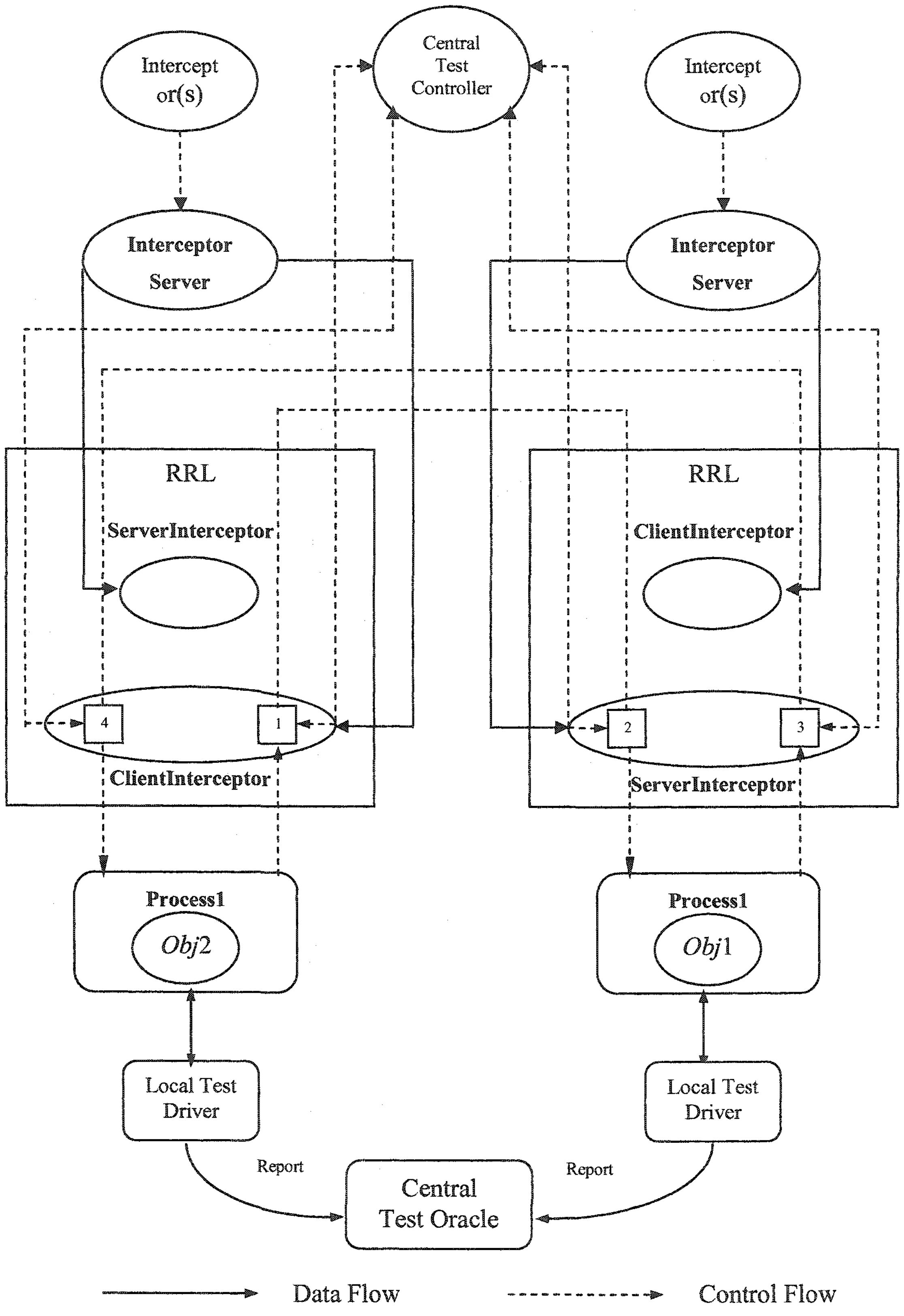


Figure 8.7: The centralized testing environment with automated control

The testing components in the centralized testing architecture work almost the same way as those we have described in the distributed architecture, except that a central test controller receives requests from all the test drivers and interceptors, and will update the program state after an event happens.

Table 8.2 shows three independent runs for each testing architecture. These experiments are conducted under the testing settings described in the introduction of this chapter, with the Online Conference as the application under test containing nine controlled events.

| | **Distributed** | **Centralized** |
|---|---|---|
| **First Run** | 3281 ms | 2156 ms |
| **Second Run** | 2938 ms | 2094 ms |
| **Third Run** | 3531 ms | 2438 ms |

Table 8.2: Performance Comparison between Distributed and Centralized Testing Architecture

These experiments show that with an AUT containing only two processes and nine controlled events, the centralized testing architecture always has a better performance than the distributed one. This is because with only few processes and test data (controlled events specified in the test scenario), the network delay of a distributed environment will dominate the overall time cost of test control. However, the distributed testing environment will have a better performance than the centralized one when involved processes and the volume of test data increase. This is because the time cost of testing an AUT is essentially decided by the input size (number of controlled events specified in the test scenario) of this AUT. The network communications will increase when the volume of test data increase, and at a certain point, the central test controller will become a bottleneck in these communications.

Now, let us consider another test scenario with the same experiment settings as those we described in the introduction of this chapter. This time, however, we increase the number of processes in the AUT to three running on three machines, with eighteen events involved in communications. The test scenario is given below:

*ie1* = (1, 0, 1, "i"),

*ie2* = (1, 1, 1, "i"),

*ie3* = (2, 0, 1, "i"),

*ie4* = (2, 1, 1, "i"),

*ie5* = (3, 0, 1, "i"),

*ie6* = (3, 1, 1, "i"),

*oe1* = (1, 0, 1, "o"),

*oe2* = (1, 1, 1, "o"),

*oe3* = (2, 0, 1, "o"),

*oe4* = (2, 1, 1, "o"),

*oe5* = (3, 0, 1, "o"),

*oe6* = (3, 1, 1, "o"),

*re1* = (1, *"OnlineConference", "permissionRequest"*, 1, *"qc"*),

*re2* = (1, *"OnlineConference", "permissionRequest"*, 2, *"qc"*),

*re3* = (2, *"OnlineConference", "permissionRequest"*, 1, *"qc"*),

*re4* = (2, *"OnlineConference", "permissionRequest"*, 2, *"qc"*),

*re5* = (3, *"OnlineConference", "permissionRequest"*, 1, *"qc"*),

*re6* = (3, *"OnlineConference", "permissionRequest"*, 2, *"qc"*)

$C=\{(ie1,ie3,ie5),(re2,re3),(re4,re5),(oe1,ie2),(oe3,ie4)\}$

$O = ((oe2, oe3) \wedge (oe4, oe5))$

Figure 8.8: A Test Scenario with three processes and 18 events

In this test scenario, we require that these three processes must speak in the order: Peer-0 $\rightarrow$ Peer-2 $\rightarrow$ Peer-3 ($\rightarrow$ denotes the relation "happen before"), which is specified in the test constraint *C*. Like in section 8.1.3, where we independently run the *Online Conference* three times, and obtain the average time cost of 3943 ms, we also run this AUT three times independently with the new test scenario, and thus obtain the time cost for each run: 5863 ms, 5938 ms and 5972 ms respectively. So, the average time cost for the *Online Conference* consisting of three processes and eighteen controlled events is: 5656 ms. Similarly, we perform an experiment with the AUT containing four processes running on four machines with twenty four controlled events involved in communications; and we require that these four processes must speak in the order: Peer-0 → Peer-2 → Peer-3 → Peer-4. Again, by independently running this AUT three times, we obtain the time cost for each run: 12085 ms, 10153 ms, and 11296 ms respectively. So the average time cost for the *Online Conference* consisting of four processes and twenty four controlled events is: 11178 ms. Table 8.3 lists the average time cost for each experiment.

| | **2 processes**<br>**9 events** | **3 processes**<br>**18 events** | **4 processes**<br>**24 events** |
|---|---|---|---|
| **Average Time** | 3943 ms | 5924 ms | 11178 ms |

Table 8.3: Time cost for running the AUT with different processes and events

By comparing the time cost for each experiment, we observe that the time cost is approximately proportional to the number of controlled events.

# 9. Conclusions and Future Work

In this thesis work, we have presented an approach to automated reproducible testing for distributed Java applications, via additional interception services into the Java RMI middleware. With the availability of the interception service, we can easily incorporate any testing environment to intercept the remote calls without modifying the AUT. Here we have outlined a code-intrusion-free testing environment with which one can gain some control over the nondeterministic choices through the predefined order among input events and remote call events. This provides support to reproduce or replay a test in concurrent and distributed systems.

We defined the format of test scenarios, discussed in detail how the interception service is injected into the RMI middleware in order to provide a mechanism to hook up testing components transparent to user applications. We also explored the use of CORBA Portable Interceptors, a similar interception technique to Java RMI interception services, and further described how to utilize this Portable Interceptors to incorporate the existing testing components, i.e., how to reuse the testing components in CORBA-based applications implemented in Java. We compared the similarities and differences between these two testing environments in a variety of aspects. We also did several experiments based on the Online Conference example to illustrate the overall testing architecture works well, and showed performance of this testing architecture based on the analysis of experimental results.

As a final remark, we would like to mention that although we tried to handle the nondeterminism, it is apparently not necessary to deterministically control every internal nondeterministic choice of the execution of an AUT. Here we have adopted the term reproducible testing in a general sense that we can control the execution over some important internal choices. Normally these important internal choices include the order of accessing shared objects and the order of remote calls. Here we have focused on the latter.

Finally, we list some limitations in the current work and possible lines of future research work:

- When a program output is passed to the central test oracle, it is passed together with the time stamp of the receipt of the output, so that the real-time related conditions can be checked. The lack of global clock in distributed systems may cause problems on the preciseness of the validation of real-time test oracles. Further investigation in this issue is on demand.
- One issue needed to be addressed in automated testing is the generation of test scenarios. In our approach, we have assumed that a set of test scenarios are given in the sense that it is feasible (see below for the meaning of test scenario feasibility). In order to automate test generation, it is necessary to analyze some formal objects, such as source code or formal specifications. Apparently, our testing approach is specification-based, so a related work is to systematically and automatically obtain test scenarios. Precisely, given formal system specifications, how do we identify and automatically generate the significant test scenarios? I am interested in searching for suitable solutions to it.
- Another challenge involved in automatic test scenario generation is the feasibility check. The feasibility check is to verify the conformance between test scenarios and a program's implementation. Thus, a test scenario that is feasible cannot cause the program to terminate abnormally lead to a deadlock/starvation state. For certain testing criteria, a significant proportion of test scenarios are infeasible in terms of the semantics of the program [37]. In the case that a given test scenario is infeasible, controlling the execution according to it may lead to concurrency related problems such as deadlock or starvation. The investigation in the feasibility of the test scenarios remains part of my future work along this line of research.
- To assure the quality of selected set of tests, we also need some test adequacy criteria, which are used to determine whether a test suite provides an adequate amount of testing for a program under test [36]. Testing adequacy analysis involves finding areas of a program not exercised by a set of tests and creating additional tests to increase testing coverage. In our approach, we assume that the given test scenario is an

*important* one, which could be used to uncover some concurrency related problems. In a real application, however, we need to develop some techniques to systematically and automatically identify the set of *important* test scenarios which is small enough to be exercised in a relatively short period of time and is sufficient enough to discover all or most of the potential faults in a program.

# Bibliography

[1] Richard H. Carver and Kuo-Chung Tai, "Replay and Testing for Concurrent Programs", volume 8, Mar. 1991. IEEE Software.

[2] H. Sohn, D. Kung, P. Hsia, Y. Toyoshima and C. Chen "Reproducible Testing for Distributed Programs", Mar. 1996. 4th International Conference on Telecommunication Systems, Modeling and Analysis, Nashville, TN.

[3] X. Cai and J. Chen, "Control of Nondeterminism in Testing Distributed Multi-Threaded Programs", 2000. Proc. of the First Asia-Pacific Conference on Quality Software (APAQS 2000).

[4] M. Ben-Ari. "Principles of Concurrent and Distributed Programming", Prentice Hall International Series in Computer Science, 1990.

[5] Peer Hasselmeyer and Marco VoB., "Monitoring component interaction in Jini federations"
<URL: http://www.ito.tu-darmstadt.de/publs/papers/itcom01.pdf >

[6] D.C. Kung Hwan Wook Sohn and Pei Hsia. "Corba components testing with perception-based state behavior", in COMPSAC'99, the 23rd Annual International Computer Software and Applications Conference, pages 116–121, 1999.

[7] Hwan Wook Sohn, David C. Kung, Pei Hsia, "State-based Reproducible Testing for CORBA Applications", International Symposium on Software Engineering for Parallel and Distributed Systems May 17 - 18, 1999

[8] Denis Reilly and A. Taleb-Bendiab, "Dynamic Instrumentation for Jini Applications", 3rd International Workshop on Software Engineering and Middleware, 2002

[9] Yingxu Wang, Grahap King, Mohamed Fayad, Dilip Patel, Lan Court, Geoff Staples, and Margaret Ross "On Built-in Test Reuse in Object-Oriented Framework Design", ACMCS115

[10] Ann Wollrath and Jim Waldo, "The Java Tutorial: RMI"
<URL: http://java.sun.com/docs/books/tutorial/rmi/index.html>

[11] Jguru, "Fundamentals of RMI: Short Course", Feb., 2002
<URL: http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html>

[12] Jeremy Blosser, "Explore the Dynamic Proxy API", Nov., 2000
<URL:
http://developer.java.sun.com/developer/technicalArticles/DataTypes/proxy/>

[13] Sun Microsystems, Inc., "Dynamic Proxy Classes"
<URL: http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html>

[14] Yingxu Wang; G. King and H. Wickburg, "A Method for Built-in Tests in Component-based Software Maintenance", 1999, 186-189, Software Maintenance and Reengineering

[15] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, "Design Patterns", 1995 Addison-Wesley

[16] OMG, "Common Object Request Broker Architecture: Core Specification", 2000
<URL: http://cgi.omg.org/docs/formal/02-11-01.pdf>

[17] Sun Microsystems, Inc., "Java™ Remote Method Invocation Specification", v1.4, 2002
<URL: ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf>

[18] Sun Microsystems, Inc., "Java IDL and Java RMI-IIOP Technologies: Using Portable Interceptors (PI)"
<URL: http://java.sun.com/j2se/1.4.1/docs/guide/idl/PI.html>

[19] Kuo-Chung Tai, Richard H. Carver, and Evelyn E. Obaid, "Debugging Concurrent Ada Programs by Deterministic Execution", Software Engineering, IEEE Transactions on, Volume: 17 Issue: 1, Jan 1991, Page(s): 45 –63

[20] Chow, T. S., "Testing Software Design Modeled by Finite-State Machines", IEEE Trans. On Software Engineering, SE-4 (3), pp. 178-187

[21] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, A. Ghedamsi, "Test Selection Based on Finite State Models", IEEE Transactions on Software Engineering, 1991, pp. 591-603

[22] Bengi Karacali, Kuo-Chung Tai, "Automated Test Sequence Generation Using Constraints for Concurrent Programs", International Symposium on Software Engineering for Parallel and Distributed Systems, May 17 - 18, 1999

[23] Richard H. Carver, Kuo-Chung Tai, "Static Analysis of Concurrent Software For Deriving Synchronization Constraints", Proc. IEEE Int'l Conf. Distributed Computing Systems, pp.544-551, May 1991

[24] Richard H. Carver and K. C. Tai, "Test Sequence Generation From Formal Specifications of Distributed Programs", ICDCS 1995: 360-367

[25] Richard H. Carver and Kuo-Chung Tai, "Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs", IEEE Transactions on Software Engineering, June 1998 (Vol. 24, No. 6), pp. 471-490

[26] Sun Microsystems, Inc., "Creating a GUI with JFC/Swing"

<URL: http://java.sun.com/docs/books/tutorial/uiswing/index.html>

[27] Sun Microsystems, Inc., "The Reflection API"
<URL: http://java.sun.com/docs/books/tutorial/reflect/index.html>

[28] D.C. Kung, N. Suchak, J. Gao, P. Hsia, "On Object State Testing", Proceedings of the 18th Annual International Computer Software & Applications Conference, IEEE Computer Society Press, pp. 222-227

[29] Vector Software, "Offers Automated Test Tools for Unit Level Testing for Ada, C, and C++ with VectorCAST"
<URL: http://www.vectors.com/whitepapers.htm>

[30] Dennis Peters and David L. Parnas, "Generating a Test Oracle From Program Documentation", Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)
<URL: http://www.engr.mun.ca/~dpeters/papers/issta.pdf>

[31] Xia Cai, Lyu, M.R., Kam-Fai Wong, Roy Ko, "Component-Based Software Engineering: Technologies, Development Frameworks, and Quality Insurance Schemes", Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific, 2000

[32] Sun Microsystems, Inc., "Java IDL Technology Documentation"
<URL: http://java.sun.com/j2se/1.4.2/docs/guide/idl/index.html>

[33] Roy Friedman, Erez Hadad, Inc., "Client-side Enhancements using Portable Interceptors" Sixth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'01), January 08 - 10, 2001 Rome, Italy

[34] R. Baldoni, C. Marchetti and L. Verde Dipartimento, "CORBA request portable interceptors: analysis and applications"
<URL: http://www.dis.uniroma1.it/~irl/pubs/concurrency02.pdf>

[35] Berard, E.V., "Issues in the testing of object-oriented software", Electro/94 International. Conference Proceedings. Combined Volumes. , 1994, Page(s): 211-219

[36] David S. Rosenblum, "Adequate Testing of Component-Based Software", Technical Report 97-34 Department of Information and Computer Science, University of California, Irvine, August 1997
<URL: http://www1.ics.uci.edu/~dsr/old-home-page/ics9734.pdf>

## Vita Auctoris

Kun Wang was born in 1975 in Taiyuan, China. He graduated from Harbin Institute of Technology, Harbin, China in 1998, where he received a Bachelor's degree in Computer Science and Engineering. He is currently a Master's candidate in the School of Computer Science at the University of Windsor and expects to graduate in fall 2003.