Distributed Unit Testing

Runtao QU, HIRANO Satoshi, Takeshi OHKAWA and Takaya KUBOTA
Information Technology Research Institute, AIST
National Institute of Advanced Industrial Science and Technology (AIST)
AIST Tsukuba Central 2, Tsukuba, Ibaraki, 305-8568, Japan
E-mail: nmg-all@aist.go.jp

Radu Nicolescu
Dept. of Computer Science, The University of Auckland, New Zealand

ABSTRACT
Distributed testing is often hard to implement. This is due to difficulties in handling heterogeneous environments, complex configurations, synchronization, error probing, result maintenance and automation in distributed testing. This paper describes a practical testing framework that permits automated distributed testing of distributed systems and applications. The framework extends the capability of JUnit to support test execution over heterogeneous environments and complex configurations.

KEY WORDS
JUnit, Distributed Application, Distributed Testing

1. Introduction

There are many types of software testing methodologies, such as unit testing, regression testing, acceptance testing, functional testing, black box testing, integration testing. However, there is another type of testing that we will focus on in this paper, distributed testing [1].

Distributed testing is sometimes confused with remote testing, which runs tests on remote computers. In most cases, remote testing is not distributed. You might run several remote tests on several remote computers, and collect all of the results back to your local computer. However, as long as these tests do not involve any interactions among the remote computers, this is remote testing rather than distributed testing.

In distributed testing, a distributed test case consists of two or more components residing in different computers which form a unified test case. These components interact with each other during the test execution. It is the interaction and interoperation between the different test case components that distinguishes distributed testing from other types of test. Fig.1 described a simple scenario for distributed testing. In Fig.1, the test programs are separated in two computers. They interact and interoperate with each other.

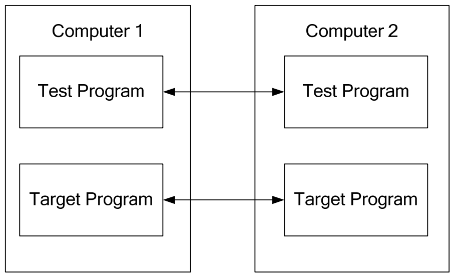


Fig. 1 Distributed Testing

For instance, distributed testing is needed for the communication middleware used in distributed applications, including Remote Procedure Call, Distributed Objects, Web Services, CORBA, etc. However, writing test code and carrying out distributed tests are difficult due to complex program structure and complex system configuration. In this paper, we describe DisUnit, an automated distributed testing framework for distributed systems including middleware and applications. In Section 2, we discuss the challenges which need to tackle in distributed testing. Section 3 presents the core framework architecture of DisUnit, which is based on JUnit. Section 4 describes the usage of the DisUnit framework. In Section 5, a case study based a middleware called HORB is presented. Section 6 discusses related research approaches. Section 7 concludes with a short summary of DisUnit.

2. The Challenges of Distributed Testing

In this section, we discuss several issues that arise in testing distributed systems including middleware and applications.

2.1 Heterogeneous Environments

Distributed systems have to work under various kinds of environments, for example, different hardware architectures, different operating systems, and various software components and versions. A good example is the

Java platform, which has many versions such as JDK1.2, JDK1.3, JDK5.0, etc. To ship a product, its vendor should test it in all combinations of supported environments. When a new environment is introduced to a distributed system, or even when some software versions are changed, the whole system has to be tested again to ensure that it conforms to the specifications.

2.2. Complex Configurations

When starting a distributed system, it is common for some setup parameters to be passed to the system. For example, omniORB, a CORBA implementation, has more than 20 parameters for ORB setup. In many cases, the application has to be started and stopped many times for testing. In other words, complex configurations induce complex testing.

2.3. Synchronization

Synchronization is an important issue in distributed testing. In a distributed system, many hosts and sometimes a database, web server, and/or application server are involved. If a remote host or server is down or slow, the testing may hang forever. On the other hand, deadlock also happens very easily in distributed systems. Setting a timeout is thus a necessity in distributed testing. Functionality of timing a test was not included in previous version of JUnit. The latest JUnit 4.0 introduces timeout as a new feature. However, JUnit 4.0 is not backward-compatible with previous Java platforms.

2.4. Error Probing

The purpose of testing is to find and locate bugs in target software. In a standalone local testing environment, it is not normally difficult to locate the error point. Combining test results with a debugging tool, developers can find the error location in the program. However, in a distributed testing environment, the programs to be tested are located on many hosts and an error could happen in any of these, and developers have to know the location of the program to locate the errors.

2.5. Result Maintenance

In typical existing testing frameworks, test results are displayed and then discarded. In many cases, only the latest results are retained. Because there is only one environment for the testing, the latest test results are often all that developers need. For instance, JUnit just runs tests and displays the results in the console or GUI. For JUnit users, it is more important to retain all tests that are passed than it is to store the results for further analysis. In distributed testing, the environments are heterogeneous; the number of combinations of environments and configurations is huge. The tests passed in one environment and/or one configuration may not work under other environments and/or configurations. Without a good result repository, developers can easily become confused by the test results. They will have difficulty remembering on which platforms a test succeeded and under which configurations it failed.

2.6 Distributed Automation

Automation is needed to enable a testing framework to run many tests at one time. However, automation in distributed testing is more difficult, due to the issues mentioned above. Distributed tests have to be started automatically in distributed heterogeneous environments with various configurations.

3. The architecture of DisUnit

To solve the above issues, we developed the following architecture for DisUnit. The framework includes six modules. Each module is designed to tackle one or more challenges in distributed testing. The overall architecture of the framework is shown in Fig. 2. The current version of DisUnit has been implemented using the Java language and is based on JUnit.

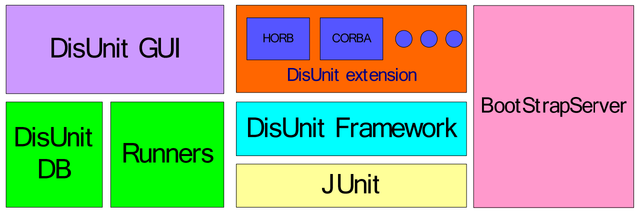


Fig. 2 DisUnit Architecture.

3.1. DisUnit Framework

This module is the core of the framework which provides the basic functionalities required in distributed testing. The purpose of this module is to help developers write distributed test programs. The class diagram of the DisUnit Framework is shown in Fig. 3.

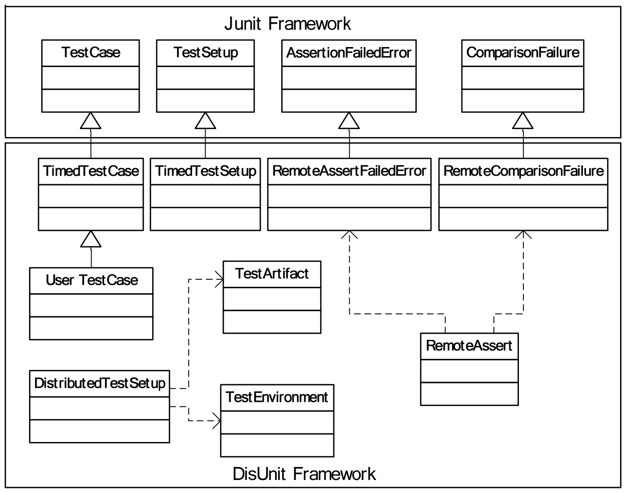


Fig. 3 Class diagram of DisUnit Framework.

The RemoteAssert class extends the capability of JUnit's Assert class to provide remote location information. The TimedTestCase class and TimedTestSetup class enable the timing of distributed tests to handle synchronization and deadlock problems in distributed testing. The TestArtifact class deals with complex configurations. The TestEnvironment class works for heterogeneous environments. The DistributedTestSetup class works with the TestEnvironment class and the TestArtifact class to deal with the automatic invocation of distributed tests. The module deals with the core issues raised in distributed testing, including heterogeneous environments, complex configurations, synchronization and error probing.

3.2. BootStrapServer

BootStrapServer is used to control the execution of distributed testing. It extends the automation capability of JUnit for distributed testing. The current BootStrapServer is implemented using HORB [3]. It supports the ApplicationInvocation interface to start and stop an application remotely for an environment and/or a configuration. It also implements the FileService interface to upload and remove files used for distributed testing.

3.3. DisUnit Extension

The framework of DisUnit is designed for extension. Although DisUnit is designed to be easy to use for all kinds of distributed systems and applications, different distributed systems and applications expose different kinds of features. The extension package is used to address this difference. The current extension implementation has only two sub packages, HORB and CORBA.

3.4. DisUnit Runners

The JUnit Framework uses runners to help developers to execute the tests. The currently popular runners in JUnit are the console based runner, the Swing based runner, and the AWT based runner. Like JUnit, DisUnit is also equipped with its own runners to execute distributed tests. In the current framework, three runners are implemented. These are an XML-based runner, a database-based runner, and a remote database runner. The XML based runner executes distributed tests and stores the results in XML files. The database-based runner executes distributed tests and sends the results to a database. The database remote runner is designed for systems which have no DB connections. In this case, the runner sends the results to a remote result server and the result server then forwards the results to a database. This is quite useful when testing embedded systems, which have no completed database interface. These specific runners assist in the automation of distributed testing as well as error probing. Another common feature of these runners are the results generated by the runners, including detailed information about the test environments and configurations.

3.5. DisUnit Database

This module is mainly used to help result maintenance. Although the results can be displayed at run-time or be recorded in XML files, the DisUnit framework is designed to use a database as its major result repository ,for easy result manipulation. The reason is that distributed testing often needs to be executed over different environments and configurations. Without a database as a result repository, developers can easily get overwhelmed by the number of results. With a database as the result repository, maintaining the test results becomes easier. The current DisUnit Database has been implemented using JDBC API. It has been tested for MS-SQL and MSDE databases. Because it utilizes the standard JDBC API, it should easily support other databases.

3.6 DisUnit GUI

This module is implemented using the Eclipse SWT. The GUI provides the capability for executing tests and querying test results using a graphical user interface. It uses a database remote runner to execute tests. A result server for result processing is embedded in the DisUnit GUI.

4. The Usage of DisUnit

4.1 Deployment of DisUnit

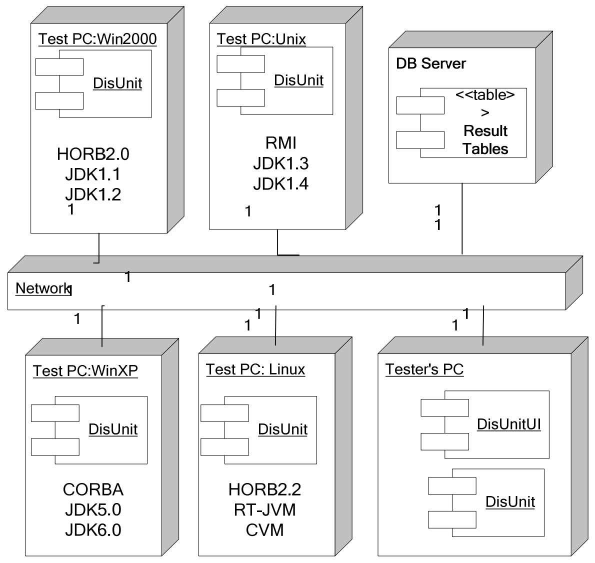


Fig. 4 The deployment diagram of DisUnit.

Fig. 4 shows a possible topology for the deployment of the DisUnit framework for distributed testing in heterogeneous environments. The environments include various OS, Java Virtual Machines and distributed middleware systems.

4.2 Write a Distributed Test

There are two scenarios for writing a distributed test. In the first scenario, only client test code is needed. In the second scenario, both client and server test code are required. A client test code calls a server test code remotely. This is common when testing for communication middleware like HORB and CORBA is to be undertaken.

Writing a DisUnit client test is as simple as writing a JUnit test. To develop a DisUnit test, the first step is to extend TimedTestCase directly, or a subclass of it from an extension package. Then a TimedTestSetup or a subclass of it should be included in the suite() method of the distributed test for server invocation. For example, the following client test code tests a simple remote server object:

```
import junit.framework.*;
import horb.orb.*;
import disunit.extensions.horb.*;
import disunit.framework.*;
public class ClientTest extends HORBTestCase{
     public void testRemoteMethod1() throws Exception{
        Server_Proxy tp = new Server_Proxy(getHorbUrl());
        assertEquals("test",tp.test001("test"));
        tp._release();
     }
   public static Test suite() {
      TestSuite suite = new TestSuite(ClientTest.class);
       HORBTestSetup horbTestSetup = new
                      HORBTestSetup(suite,"test.conf");
      return horbTestSetup;
   }
}
```

From the code, one can see that the test method is almost the same as in a JUnit test. The biggest difference is that a HorbTestSetup object is used to wrap the suite for distributed execution, and that it depends on a property file to set up the remote server for testing. The property file is used to describe the location, configuration and environment setting of a server process.

In the second scenario, DisUnit also allows developers to add an assert to a server object to help the developers to isolate the error location. This feature is useful to test communication middleware like CORBA and HORB. The following code is an example of adding a remote assert in a server object of HORB:

```
import horb.orb.*;
import disunit.framework.*;
public class Server{
     public String test001(String msg) throws Exception{
        RemoteAssert.assertEquals("test", msg);
        return msg;
     }
}
```

The failure will be caught by the runner on the client site. By using DisUnit Runner, one can run the test and generate an XML file to store the results, or send the results to the database. The test server will be invoked automatically; the required resource including test target program of the test and the configuration files will be uploaded to the server automatically. After the test, the server will be shut down automatically and all resources will be removed to keep a clean environment.

4.3 Run DisUnit Tests with DisUnit GUI

Fig. 5 shows the SWT GUI for executing tests. To run tests using the DisUnit GUI, the users need choose the OS, JRE (Java Runtime Environment) versions, etc. Then a test set is composed based on the choices. After that, users can run the test set with a single click. In the background, the DisUnit framework uses the DisUnit runner to execute selected tests for the entire combination of environments and sends the results to a database.

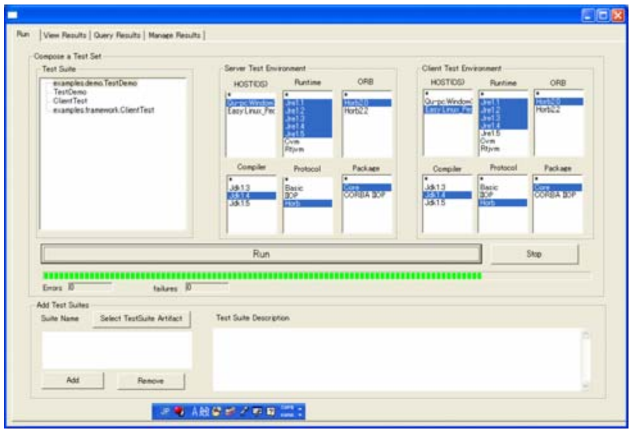
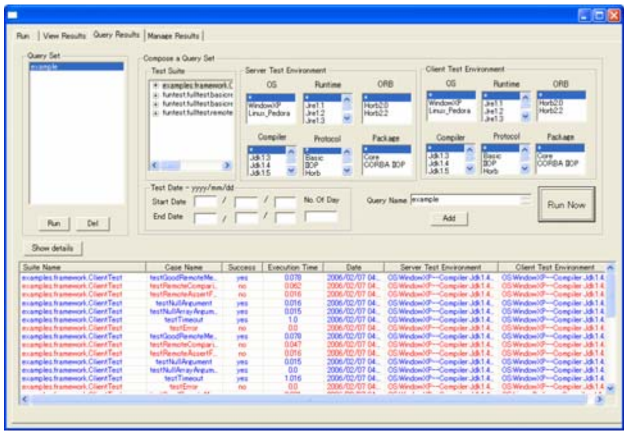


Fig. 5 The GUI to execute Tests.

For example, you can choose one test suite, a Windows machine and JRE1.1 to JRE1.5 as the Server Environment, and a Linux machine and JRE1.1 to JRE1.4 as the client environment, then click on run. The combination of all environments will be tested with this single mouse click. Information on how many errors and failures happened is displayed at run-time. After the whole test execution finishes, the GUI will automatically switch to the result page to display the result summary. You can then explore the detailed results.

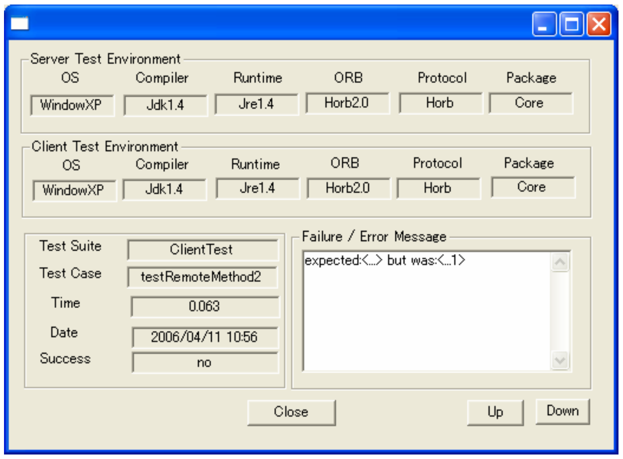
4.4 Query Test Results with DisUnit GUI

Another important role of DisUnit GUI is to provide a query interface to the test results. Fig. 6 shows the SWT GUI for querying test results. With DisUnit GUI, you can compose a query set, which can query results based on test suite name, test environments, date and combinations of all of these. You can then explore the test results by selecting a result. Fig. 7 shows a single test result.



Suite Name	Case Name	Success	Execution Time	Date	Server Test Environment	Client Test Environment
examples.framework.ClientTest	testGoodRemoteMe...	yes	0.078	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testRemoteCompari...	no	0.062	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testRemoteAssertF...	no	0.016	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testNullArgument	yes	0.016	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testNullArrayArgum...	yes	0.015	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testTimeout	yes	1.0	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testError	no	0.0	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testGoodRemoteMe...	yes	0.078	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testRemoteCompari...	no	0.047	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testRemoteAssertF...	no	0.016	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testNullArgument	yes	0.015	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testNullArrayArgum...	yes	0.0	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testTimeout	yes	1.016	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4
examples.framework.ClientTest	testError	no	0.0	2006/02/07 04...	OS:WindowXP--Compiler:Jdk1.4...	OS:WindowXP--Compiler:Jdk1.4

Fig. 6 The GUI to query results.



Server Test Environment

OS	Compiler	Runtime	ORB	Protocol	Package
WindowXP	Jdk1.4	Jre1.4	Horb2.0	Horb	Core

Client Test Environment

OS	Compiler	Runtime	ORB	Protocol	Package
WindowXP	Jdk1.4	Jre1.4	Horb2.0	Horb	Core

Test Suite	ClientTest
Test Case	testRemoteMethod2
Time	0.063
Date	2006/04/11 10:56
Success	no

Failure / Error Message

```
expected:<...> but was:<...1>
```

Fig. 7 The Detailed Test Result.

4.5 The compatibility with JUnit

We have tried to keep the DisUnit framework compatible with JUnit. The distributed tests can even be executed by a JUnit runner. You also can execute the distributed tests using the JUnit runner for Apache Ant or Eclipse.

5. Case Study

A network middleware called HORB [3] was used to carry out the case study for the DisUnit platform. HORB is a lightweight pure-Java ORB that is designed to provide high performance for distributed computing. It has been applied to a wide variety of systems including stocking trading system, robotics and manufacturing.

While HORB offers many useful features, such as connection reuse, threading pooling, synchronous and asynchronous method calls, and dynamic object activation which are found in CORBA products and RMI (Remote Method Invocation), a careful design enabled it to gain higher performance. With HORB, Java code can be programmed naturally, and remote objects (POJOs) do not need to inherit an interface or implement a specific class. This is a very important feature of HORB for building an object system using design patterns or for migrating legacy code for distributed computing.

HORB is an open source project that started 10 years ago. At that time, XP programming was not popular and JUnit had not been invented. The original HORB source code came without full testing. The quality of HORB has been unknown to its developers since its inception. This situation prevents many developers from adopting HORB in their projects. Using DisUnit, we developed a full functionality test package for HORB. The test package includes 32 test suites and 492 test cases.

With the help of the DisUnit framework and HORB extension, the procedure to setup and shutdown remote HORB servers is handled by DisUnit and its BootStrapServer. DisUnit also removes the configuration information from the test code, which permits the test to run in different environments and configurations without having to edit the code. So if developers need to test a distributed application over multiple platforms, DisUnit is a good tool to save time and reduce cost.

In the case of the HORB test suites, we had to run them on Sun's JDK platform, from JRE1.1 to JRE1.5 and their combinations, across Windows and Linux machines in both client and server sites. For example, a server might run in the Windows JRE1.1 environment and a client might run in the Linux JRE1.5 environment. The total number of combinations is 100. With DisUnit, you can finish the testing over the entire number of possible combinations with a single mouse click.

Using DisUnit, more than 20 bugs were found in HORB, the quality of HORB was substantially improved. We also found some compatibility problems to run HORB on different Java platforms. Table 1 lists some of the results. In Table 1, the tests were performed on HORB compiled with Sun's JDK1.4 and targeting JDK1.1. In theory, this should work for all combinations of JDK platforms. However, we found the results shown in Table 1 instead. As can be seen, many combinations failed.

Table 1. Interoperability test of HORB across different JRE

Client\ Server	JRE1.1	1.2	1.3	1.4	1.5
1.1	ok	fail	fail	fail	Fail
1.2	fail	fail	fail	fail	Fail
1.3	ok	fail	ok	ok	ok
1.4	ok	fail	ok	ok	ok
1.5	ok	fail	ok	ok	ok

As illustrated in the last section, DisUnit should be quite easy to learn. An experienced JUnit user can learn DisUnit within a couple of hours. We thus believe that the DisUnit framework could substantially increase the efficiency of distributed testing.

6. Related Research

There is much research targeting the testing of distributed systems. However, most research targets only particular distributed systems. In paper [2], an automated distributed test environment for RTI (Run Time Infrastructure) component is constructed. However, this test environment is not applicable to other distributed systems. Paper [4] proposed a Scenario-Based Test Framework for testing distributed systems. This paper focuses on how to generate test programs. Tempto is a test framework and test management infrastructure based on Fine-Grained XML-Documents [5]. The testing infrastructure is inherently distributed. However, it only focuses on test and result management. IBM's Rational Functional Tester [7] and Eclipse's TPTP [8] are good tools for remote testing of applications and testing of web applications. However, they do not support distributed testing in nature. Special approach is needed to adopt them for distributed testing.

JUnit, developed by Kent Beck and Erich Gamma, is the de facto standard unit testing library for the Java language. JUnit kick-started and then fuelled the testing explosion [6]. JUnit (itself inspired by Smalltalk's SUnit) has inspired a whole family of xUnit tools including nUnit (.NET), pyUnit (Python), CppUnit (C++), and dUnit (Delphi).

JUnit has many extensions for different kinds of applications. Some well-known ones are dbUnit for database applications; HttpUnit for web server testing; and Cactus for Servlet testing. Except for that, JUnit has broad out-of-the-box support in products like Apache Ant, Maven and Eclipse.

DisUnit is an extension of JUnit for distributed systems and applications. While there are some extensions of JUnit for thin client distributed systems such as web applications (HttpUnit) and Servlets (Cactus), there is a need for a testing platform which can be used for testing rich client distributed systems. DisUnit has been developed for this purpose.

7. Summary and Future Work

DisUnit has been designed to provide ease of use for the distributed test developers. The effectiveness of the framework is shown by testing a distributed middleware called HORB.

The current version of DisUnit only works with JUnit 3.8 and it is not compatible with JUnit 4.0. The reason is that DisUnit will be used for all Java platforms including Sun's Java platforms and other Java products. JUnit 4.0 is not in the least bit backwards-compatible with previous Java Platforms.

The next step for DisUnit is to improve its ease of use for distributed test developers by developing Apache Ant, Maven and Eclipse plug-ins for DisUnit and to provide some more extension packages. We are also working with the Department of Computer Science, University of Auckland, New Zealand to create domain specific languages (DSL) [9] to support the distributed testing and performance benchmarking for distributed systems.

Testing is difficult and expensive. However, with JUnit, programmers "love" testing [6]. Testing distributed systems and applications is even more difficult. We believe DisUnit can relieve programmers and make them "love" distributed testing. A trial version of DisUnit will be available soon through the HORB website horb.a02.aist.go.jp.

Acknowledgement
We thank Ulrich Speidel for comments which improved our presentation.

References

[1] Managing Distributed Testing, TETware White Paper, http://tetworks.opengroup.org/Wpapers/distributed_white paper.htm.
[2] John Tufarolo, Jeff Nielsen, et al., Automated Distributed System testing: Designing an RTI verification system, Winter Simulation Conference, 1094-1102, 1999.
[3] HIRANO, S., HORB: Distributed Execution of Java Programs, Lecture Notes in Computer Science 1274 pp.29-42, 1997.
[4] Wei-Tek Tsai, Lian Yu, Akihiro Saimi, Scenario-Based Object-Oriented Test Frameworks for testing Distributed Systems, 9th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003), 288-294, 2003.
[5] Ralf-Dieter Schimkat, Martin Schmidt-Dannert, Wolfgang Küchlin, Rainer Krautter, NetObjectDays, Tempto-An Object-Oriented Test Framework And Test Management Infrastructure Based On Fine-Grained XML-Documents, In NetObjectDays 2000 -Object-Oriented Software Systems, pages 274–287, Erfurt, Germany, Oct. 2000.
[6] Elliotte Harold, An early look at JUnit 4, IBM's developerWorks Java technology, www.ibm.com/developerworks/java/library/j-JUnit4.html, 2006.
[7] IBM Rational Functional Tester, http://www.ibm.com/software/awdtools/tester/functional/
[8] Eclipse Test and Performance Project home page, http://www.eclipse.org/tptp/.
[9] Special issue on domain-specific languages. IEEE Transactions on Software Engineering, 25(3), May/June 1999.