Case Study of Testing a Distributed Internet-System

by

DI Siegfried GOESCHL
IT Serv GmbH, A-1120 Vienna, AUSTRIA
Email: goeschl_s@excite.com

&

Harry M. SNEED
Software Daten Service, A-1200 Vienna, AUSTRIA
Email: harry.sneed@t-online.de

Abstract
This paper describes the testing strategy, methods and tools used for testing a distributed financial services system on the Internet with CORBA interfaces for handling a high volume of queries to a relational database. The new system architecture was derived by reengineering a previous more limited application. The paper starts with an outline of the software architecture, the testing requirements and the testing strategy. It then goes on to describe the tools used in the project and the results achieved. For the project a generic C++ integration test framework was developed especially for testing distributed components. This tool is described in detail. The paper ends with a discussion of the discovered defects and their distribution.

Keywords: Unit Testing, API Testing, Interface Test Drivers, Message Generation, Interface Validation, Regression Testing
1 Introduction

On the one hand, there is a general lack of adequate tools for testing distributed systems. On the other hand, there is a great need for such tools, especially in light of the effort required for testing such systems. It has been noted that testing takes at least one half of the effort required to develop object-oriented, distributed systems. [Binder94a]

The special problems involved in verifying and validating distributed software have been addressed in the appropriate literature [Bourne97]. There are not only multiple nodes of a network with an exponential number of interconnections, but also multiple architectural layers to be tested. This raises test costs to an even higher level. In addition, there are four distinct levels of testing to be done:

- Module or class testing, often done as White-Box testing,
- Integration testing, often done as Grey-Box testing,
- System testing, often done as Black-Box testing, and
- Acceptance testing

The testing of individual classes can be handled with the help of existing commercial unit test tools. System testing can also be done using conventional capture/replay and database editing tools [Beizer95]. Acceptance testing represents a greater challenge in the case of distributed systems, but it is not critical if the other tests have been performed well. The greatest challenge is that of integration testing. Distributed systems consist of a large number of interacting components executing in diverse environments with complex interfaces. These interactions and their interfaces have to be verified and validated by integration testing since neither unit nor system testing suffices for this purpose. The only way to accomplish this in an effective and efficient manner is via automated tests. As pointed out by Graham and Fewster [FewGra99], test tools may not replace human intelligence in testing, but without them testing complex systems at a reasonable cost will never be possible.

The tool described in this paper grew out of an internet project in Great Britain in the years 1998/99. The project combined web servers with distributed CORBA servers (Common Object Request Broker Architecture) and relational databases. The goal of the project was to provide clients in different parts of the world with rapid access to financial data. The standard queries needed to be processed within seconds on relational tables with several million entries.

The project in question was actually a reengineering project since the basic application had already existed in the form of two different server systems providing access to a Sybase database. One of these server systems was programmed in Standard C, the other in the 4th Generation language FORTE®. The results of the queries were delivered as SGML formats (Standard Generalized Markup Language). However, this original solution had reached its limits. It was becoming increasingly slow and unreliable under the increasing load of the clients. Therefore, as often happens with quick-and-dirty solutions, management decided to reengineer the system, re-implementing the critical transactions in C++ with a new CORBA server for faster message distribution and a separate SGML front-end to transform the data into SGML. At the same time the web server implementation and the database layout were reengineered to increase the throughput and maintainability of application by independent teams located in London and Washington.
The project team chose a staged delivery lifecycle strategy with the option to use the existing server implementation as a fallback. In the case of catastrophic production problems the new CORBA-based servers could have been activated or deactivated by modifying a single configuration file. The resulting architecture and concurrent reengineering tasks are depicted in Figure 1.

![Architecture Overview](image)

**Figure 1 - Architecture Overview**

The web server receives HTTP (*Hyper Text Transport Protocol*) requests, converts them to the SGML requests and dispatches them to the appropriate server depending on the request types defined by a configuration file. The SGML server accepts SGML requests from the web server and invokes one or more IDL functions (*Interface Definition Language*) on the CORBA Server using IIOP (*Interoperable Internet Object Protocol*). The CORBA server delegates the calls to a database layer which encapsulates the business logic and shields the server from changes to the database layout.

The reengineering project was subject to the following constraints:

- the project team had little experience using a SYBASE® database
- the functionality to be reengineered in the first development increment was either written in C or FORTE® and consisted of approximately 10000 lines of legacy code
- the original database also had to be reengineered due to performance problems
- the web server implementation was rewritten in order to use ASP (Active Server Pages) instead of in-process PERL scripts (i.e. Fast-CGI)
- the existing monolithic, single-entry API (Application Programming Interface) had to be replaced by a modular IDL interface to enable reuse
- developing a CORBA server in C++ is error-prone since the developer bears responsibility for the memory management
- the resulting product had to support multiple platforms (SOLARIS and Windows NT)

In view of these constraints, testing was assigned a particularly high value from the start. It was decided to commit at least half of the project resources to testing, to plan the test carefully, to make a test plan in accordance with prevailing standards, and to develop appropriate test tools for executing the tests. The project should be test driven in accordance with the recommendations of Hetzel, Beizer, Binder and others [Hetzel88],
[Beizer95], [Binder99]. It was mainly due to this emphasis on testing that the project came to a successful conclusion. Without the aid of automated test tools embedded in a well-defined test process as described by Koomen and Pohl [KoomenPol99], the project could never have been completed on time.

## 2 Test Strategy

In planning the testing, the following requirements had to be considered:

- ensure that the SGML server is fully backward compatible with the existing C and FORTE server implementation
- ensure that the new system has a better average response
- ensure that the new system is reliable and stable under high load
- ensure that the tests can be run on multiple platforms

Altogether, there were three layers of software to be tested first individually and then in conjunction with one another.

- SGML Server
- CORBA Server
- Database Access Layer

Using a bottom-up test strategy, the architectural layers were to be tested in reverse order, starting with the database access layer and working up to the client as shown in Figure 2.

![Bottom Up Testing Strategy Diagram](image)

Figure 2 - Bottom Up Testing Strategy

### 2.1 Database Access Layer Test

To test the database access layer it was decided to create a regression test suite written in C++ to compensate for the lack of SYBASE experience and to test the changes in the database schema. This regression test also provided a way to benchmark various queries
against the database. Since there were multiple ways of accessing the same data, this was important for recording and comparing results.

### 2.2 CORBA Server Test

To test the CORBA server, it was decided to implement a proxy client application to generate requests to the server. This proxy client should simulate the behavior of a later real client in generating all kinds of requests in any order. The focus for this test was to prove the absence of memory leaks and to test the reliability of the ORB (Object Request Broker) under high load.

### 2.3 SGML Server Test

To test the SGML Server it was decided to create a regression test suite and run it with manually created requests in accordance with the API specification as well as with hundreds of real-life requests from the production system. The requests extracted from production served to represent the existing operational profile. The requests derived from the specification served to represent the existing and new requirements.

### 3 Test Toolbox

To facilitate testing, a set of tools was used consisting of:

- **Assertion Package** – an industrial strength assertion package
- **Logfile Viewer** – a tool for monitoring multiple logfiles
- **C++ Unit Test Framework** – a unit and integration test framework for C++
- **PERL Regression Test Framework** – an integration and system test framework

Each of these tools will now be considered in turn.

### 3.1 Assertion Package

Meyer’s Design by Contract [Mey97] is an important concept for software developers and reduces the time spent during debugging significantly. It requires the programmers to formulate preconditions on the states their inputs should have when they start and postconditions on the states their outputs should have when they are finished. Besides, invariant assertions can be postulated on the state of objects, which should not change.

The standard assertion function in the C Runtime Library is not useful in a production environment because it prints an error message and terminates the program, which is exactly not the behavior you would like to have in a client or server test. Imagine a GUI application where the assertion function prints an error message to an invisible error console or a server which suddenly terminates. The standard assertion function is only intended for local module testing.
For this reason, it was decided to implement an assertion package with the following functionality:

- *Just In Time Debugging* Support on Windows NT
- customizable behavior
- selective activation of the assertions.

### 3.1.1 Just In Time Debugging

The idea of just in time debugging is borrowed from the Microsoft Foundation Class\(^1\) (MFC) assertion package, which starts a debugger whenever a failed assertion is encountered. It has turned out that this feature is extremely valuable for debugging and testing. It is important to know whether one’s program terminated because of some addressing error or because of a violated invariant. In the latter case, one can investigate the call stack to pinpoint the reason why the invariant was violated.

### 3.1.2 Customizable Behavior

An industrial strength assertion package has to be flexible to meet the following requirements:

- log an error message
- send an SNMP (Simple Network Management Protocol) alert to a management console
- send a mail to the system administrator
- throw an exception instead of terminating the program.

This flexibility could be achieved by implementing a Chain of Responsibility [GoF95] where the system under test can register various assertion handlers. Each assertion handler is responsible for providing a certain level of functionality before it delegates the assertion to the next handler.

### 3.2 Logfile Viewer

Under Windows NT the editors used to view a logfile have several shortcomings:

- some programs try to lock the logfile exclusively
- hardly any program updates the view of the logfile automatically
- one needs a filter to view large logfiles
- one may need to view multiple logfiles simultaneously

Therefore, it was necessary to develop a customised logfile viewer explicitly for testing client/server type applications which would not have such limitations.

---

\(^1\) MFC is a C++ GUI framework for the Windows environment
3.3 C++ Unit Test Framework

Experience has proven that it pays off to create and maintain formal unit tests as part of the software development cycle [Wie96]. Usually a software developer writes a test driver for his/her unit of work, but that test driver only ensures that the software doesn’t crash during the first execution. Typically it consists of a “main” routine and a few statements to print the results. Later the test driver is neglected so that it no longer fits with the unit under test and might not even compile. At this stage everybody has abandoned unit testing and it becomes costly to maintain the software because there is no cheap way to reproduce individual unit tests.

This dilemma is well known and the question always arises why software developers don’t create a proper unit test. Usually there are several answers:

• “it’s too difficult to test this unit separately ...”
• “there is not enough code to justify a unit test ...”
• “I don’t have enough time ...”
• “I don’t know how to write a unit test ...”

The point to be stressed here is that all of these objections are valid to a certain degree since most software developers are reasonable persons. The answers do no longer make sense if a unit test framework fulfills the following requirements:

• there is a unit test framework available
• it takes as much time to create a proper unit test as to “hack” a throw-away test driver
• one gets more “bang for the buck”\(^2\) as a software developer and this makes it more difficult to neglect unit testing

The advantages of using a reusable unit test framework instead of a throw-away test driver are obvious. The test framework supports:

• detection of memory leaks
• performance measurements
• stress testing
• test of multi-thread safety
• simple modification of test data
• integration of regression testing
• simple scripting

For these reasons and others, the project team decided to implement its own unit test framework for C++ to satisfy the special requirements of a distributed test. The result was the tool called APITEST. In the end, it was used in several projects both on SOLARIS and Windows NT.

3.3.1 Basic Concepts

Every unit test can be divided into one or more test suites with a unique name. Every test suite can contain one or more test cases, which is represented in Figure 3. Every test case is uniquely identified through a number within a test suite.

\(^2\) “Get more bang for the buck” was the manager’s original statement about making a process more effective
An invocation of a particular test case can lead to one of the following results:

- passed
- failed
- crashed
- obsolete

A test case fails if it does not fulfill all required post conditions of the test. A test case crashes if an exception is propagated out of the test case implementation. A test case is obsolete if it is not used any longer [Sneed98].

The command line interface allows one to run a unit test in multiple ways:

- run all test suites
- run one particular test suite
- run one test case within a test suite

After execution of a unit test an execution summary is displayed. A typical example is given in Figure 4.

```
<table>
<thead>
<tr>
<th>Execution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test started      : Tue Oct 05 13:39:26 1999</td>
</tr>
<tr>
<td>Test finished     : Tue Oct 05 13:39:26 1999</td>
</tr>
<tr>
<td>Test Executed     : 6</td>
</tr>
<tr>
<td>Test Crashed      : 0</td>
</tr>
<tr>
<td>Test Failed       : 4</td>
</tr>
<tr>
<td>Test Passed       : 1</td>
</tr>
<tr>
<td>Test Obsolete     : 1</td>
</tr>
<tr>
<td>TEST 3 FAILED</td>
</tr>
<tr>
<td>CPU time (sec)    : 1.022000</td>
</tr>
</tbody>
</table>
```

Figure 4 - Test Report of APITEST

### 3.3.2 Detection of Memory Leaks

One of the most difficult aspects of the programming language C/C++ when making large-scale software development is manual memory management. Therefore it is essential to simplify the detection of memory leaks.
Figure 5 - Integrating APITEST with Purify

APITEST provides close integration with PURIFY® from Rational (see Figure 5), a commonly used tool for detecting memory leaks and related problems during development. PURIFY tracks all memory and checks allocations of memory during execution, but it turned out that the results were difficult to interpret due to the way resources were allocated by initialization routines. It became necessary to execute test cases repeatedly in order to obtain sensible results. APITEST prints status messages on the PURIFY console before executing a test case and reports the allocated resources after executing the test case.

In the absence of PURIFY it is possible to track resource consumption manually, but it requires more time to detect small memory leaks. Therefore the combination of APITEST and PURIFY was determined to be the best solution.

3.3.3 Performance Measurements

The execution summary provides information about the time spent executing the test cases. For meaningful results it is necessary to execute tests more than once, otherwise results are distorted by the initialization time.

3.3.4 Stress Testing

For the server development it is also essential to test the system under full load and/or gather performance data with an increasing load. Assuming that there is a unit test acting as a client it is convenient to reuse the unit test and test the server with multiple clients over a long time. APITEST facilitates this by providing command line options to execute the tests multiple times with one or more threads. Furthermore a user-defined wait between executing two test cases is supported to simulate the behavior of real clients.
3.3.5 Test of Multithread Safety

Executing a unit test with multiple threads is important for ensuring that the unit under test doesn't deadlock itself after the first execution. Usually these tests are omitted since writing multithreaded test programs is significantly more work. APITEST facilitates these tests by its ability to define the number of executing threads on the command line. In this case the unit test is executed by multiple threads simultaneously. Furthermore, it is possible to randomize the order in which test cases are executed for every thread to simulate the behavior of real clients. Passing this test does not guarantee that the unit under test is multithread-safe, but it does give a good indication that it performs properly in a multi-threaded environment.

3.3.6 Simple Modification of Test Data

A throw-away test driver tends to hardcode the required test data, thus complicating the migration to a new test environment, e.g. a new database for integration testing. To avoid this complication it is necessary to provide an easy way to access test data. This is accomplished by providing configuration file support as shown for example in Figure 6.

```plaintext
$dbtest.1.branch_code=14000
$dbtest.1.customer_id=730875734
```

Figure 6 - test data configuration

Each entry in the configuration file consists of a key and a value. The key consists of the test suite name, the test case number and the name of the variable.

3.3.7 Integration of Regression Testing

When testing a module for accessing a database, a unit test typically doesn't verify the result apart from checking certain assumptions (e.g. there must be some results or a certain field can't be null). APITEST provides the notion of a verbose mode to produce detailed output either written to the console or into a file. This verbose output can be captured and explicitly compared with a reference to detect any differences between test runs (see 3.4. Perl Regression Test Framework later)

Additionally APITEST enables each worker thread to use its own output file which allows for a further check of multi-thread safety. Each worker thread executes the same tests and should produce the exactly the same output file. Any difference of the resulting output might indicate a race condition.

3.3.9. Simple Scripting

There will always be a couple of broken tests which do not need to be executed every time, e.g. a quick regression test run by a developer after code changes. Specifying all test cases and/or test suites which should be executed by a script provides a solution. The script file is then driven by the command line options.
3.3.10. Command Line Options

The unit test framework supports the following command line options as shown in Table 1.

<table>
<thead>
<tr>
<th>Command Line Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-C file</td>
<td>Name of test configuration file</td>
</tr>
<tr>
<td>-F file</td>
<td>Script file with tests to execute</td>
</tr>
<tr>
<td>-I</td>
<td>Ignore failed tests</td>
</tr>
<tr>
<td>-M threads</td>
<td>Number of executing threads</td>
</tr>
<tr>
<td>-R</td>
<td>Random shuffle test sequence for each thread</td>
</tr>
<tr>
<td>-S testsuite [-T testcase]]</td>
<td>Test(s) to execute</td>
</tr>
<tr>
<td>-O filename</td>
<td>Name of output file</td>
</tr>
<tr>
<td>-V</td>
<td>Enable verbose output</td>
</tr>
<tr>
<td>-w ms</td>
<td>Time to wait between two test cases</td>
</tr>
<tr>
<td>-X repeats</td>
<td>Number of repetitions</td>
</tr>
</tbody>
</table>

Table 1 - Command line options for APITEST

3.4. Perl Regression Test Framework

DROPTEST is a Perl regression test framework developed by Wild [Wild97]. It has been used in slightly modified form in many projects under Windows 95, Windows NT, SINIX, AIX and SOLARIS.

It was decided to implement a regression test suite for the SGML Server to prove that its behavior was equivalent to that of the replaced components. One part of the regression test suite was drafted according to the specification. The other part consisted of more than 200 real-life requests from the production system to cover any unspecified behavior.

3.4.2. Basic Concepts

The functionality of DROPTEST is based on the directory structure shown in Figure 7.

![DROPTEST directory structure](image-url)
The main idea is to drop a regression test into the **TEST** directory (hence the name DROPTEST) The Perl script checks every file found in **TEST** if there is a corresponding file in **PRE** and/or **POST** and executes them in the following order:

- **PRE** step (optional)
- **TEST** step (mandatory)
- **POST** step (optional)

In this manner the script provides an optional pre and post step, which can perform the following tasks:

- copy configuration information
- delete temporary files
- kill left-over processes

The basic idea is that every test step produces some sort of reproducible output. A test step is considered successful if the current output is identical to a reference output stored in the reference directory. A test step is regarded as unsuccessful if the current output is different from the reference output.

### 3.4.3. SGML Server Regression Testing

The regression test was executed with a copy of the real production database and was updated daily. Under these circumstances the regression test depended heavily on the ease of creating new reference data. This was accomplished by running the regression test against the original server written in C or FORTE to generate the reference data.

The SGML Server was tested with a PERL Script sending SGML requests to the SGML Server. This PERL script was invoked as the actual test step of DROPTEST and read an input data file as shown in Figure 8.

```
REQUEST=COMBINED_SEARCH
USERID=ROWJ
COM_FILER_NAME_SEGMENT=LEPCO
COM_COMPANY_NR=L420000000
OUT_SORT_KEY_2=DOC_TYPE
OUT_TYPE=D
OUT_SORT_ORDER_1=D
FILTER_INSIDER_DOCS=Y
OUT_SORT_ORDER_2=A
SEC_ONLY=N
OUT_START_ROW=1
OUT_TOTAL_ROWS=Y
OUT_MAX_ROWS=25
OUT_SORT_KEY_1=DOC_DATETIME_ARRIV_MIN
```

Figure 8 - SGML request input data for retrieving financial documents

In the post step caching related tags were removed from the result and compared with the reference files, by means of such automated comparison, deviations in the new results could be readily detected [Harrold98].
4 Defect Distribution

During the first development increment and a four month testing period 4700 statements of newly developed code and 7400 statements of reused in-house code (such as a CORBA-based application framework) were tested and 151 defects were found. The defect rate in the newly written part of the application is at least 10 times higher than in the reused code as shown in Table 2, a finding confirmed by Basili et al. [Basili+96]. The defects found in the reused application framework resulted from a different usage profile, which was not covered by the existing regression test suite. This effect is described as the pesticide paradox by Beizer [Beizer90] because some residual bugs are not revealed by the existing regression test suite but triggered by the new application instantiation.

<table>
<thead>
<tr>
<th>LOC</th>
<th>Defect</th>
<th>Defect/KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4700 LOC new code</td>
<td>132</td>
<td>28.1</td>
</tr>
<tr>
<td>7400 LOC reused code</td>
<td>19</td>
<td>2.6</td>
</tr>
</tbody>
</table>

Table 2 - Defects found in newly written and reused C++ code

The diagram depicted in Figure 9 shows the defects classified by severity, and tools such as DROPTEST, APITEST, build process, manual testing and reviewing source code. DROPTEST proved to be very effective due to the ease of creating new test cases by reusing existing test data.

![Figure 9 - Defects per Tool and Severity](image)

Furthermore the encountered defects could be classified as memory related problems (memory leaks, memory access violations, using already freed memory), implementation related problems and SGML related problems (formatting the SGML replies, missing SGML tags) as depicted in Figure 10.

---

3 The generated code accessing the database tables (DB Compatibility Layer) was not counted.
Figure 10 - Defects per Tool and Defect Type

APITEST was invaluable for detecting memory related defects - 63% of all memory related defects were discovered using APITEST such as:

- exception handling code generated by the IDL compiler
- memory leak in the SYBASE libraries
- memory leak and a multithread-safety problem in the CORBA naming service
- memory leak in the SNMP library

The independent system test group in Washington found three defects which were traced back to a slightly different system configuration regarding the SOLARIS operating system and the installed ORB.

5 Conclusion

Testing often becomes the critical-path activity on a software project because it is postponed to the end of the software development process. With a test driven development approach, unit and integration tests are developed in parallel with the software and considered as part of the software release. If developers create and maintain tests for their work, the software will never be allowed to disintegrate. Consequently these tests are used for a Daily Build and Smoke Test [McCon96] in which a software product is completely built every day and then put through a series of tests to verify its basic operations.

The test driven development approach proved to be highly effective during the incremental development:

- the assertion package with Just-In-Debugging allowed pinpointing violated assertions immediately using an interactive debugger
- 63% of the memory related defects including memory leaks in commercial software packages were found by using APITEST in conjunction with PURIFY
- using DROPTEST with SGML test data saved considerable effort by integrating real-life requests from the production system and by reusing test data to create new test cases
- the ease of generating an arbitrary load with APITEST under various conditions encourages developers to do performance testing of components at a very early stage of development which in turn increases the reliability of the final product.
The layered testing approach is essential for efficient software correction in a distributed system. By "drilling down" the various layers of the software it is possible to pinpoint problems and to debug the offending code very efficiently. The "drilling down" starts with a failed integration or system test case and involves executing, reviewing and finally creating test cases for the software layers beneath. The intention is to pinpoint the problem and to improve the existing test suites because a defect detected during integration testing indicates a missing unit test case.

All in all, it can be concluded that the use of a unit and integration test framework to detect errors in the components and their interfaces as soon as the code was compiled was a great advantage and contributed significantly to the low number of defects found in system testing where they are much more difficult to locate and remove. However, this required that the test be designed into the system architecture from the start. This design for testability proved to be a critical success factor in the overall project. [Binder94b].
References


