The following paper was originally published in the
Proceedings of the Sixth Annual Tcl/Tk Workshop
San Diego, California, September 14–18, 1998

A Tcl-based Multithreaded Test Harness

Paul Amaranth
Aurora Group, Inc.

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

A Tcl-based Multithreaded Test Harness

Paul Amaranth
Aurora Group, Inc.
email: paul@auroragrp.com

Abstract

This paper describes an implementation of a test harness written in Tcl and C for Merit Network, Inc. capable of running multiple structured tests simultaneously. Many available test tools are based on a single threaded stimulus/response approach. This is not always sufficient to adequately test certain classes of applications that manage multiple simultaneous requests. The design presented in this paper is for a Tcl-based system capable of running multiple simultaneous tests in the context of testing a RADIUS authentication server. The design and implementation illustrate novel applications of Tcl as well as the rapid development and code reusability inherent in using Tcl as an application glue language.

1 Introduction

Merit Network, Inc is a nonprofit entity affiliated with the state universities of Michigan that is responsible for supplying Internet services to its member universities and affiliates. As part of fulfilling that mission, Merit Network, Inc. has developed an implementation of an Authentication, Authorization and Accounting (AAA) server based on the RADIUS protocol. Through the free and licensed distributions of their server, as well as participation in the IETF and RADIUS working groups, Merit has been a driving force in the development of the RADIUS protocol. Recently, Merit has been the key player behind the formation of a RADIUS industry consortium formed to foster industry cooperation and shared development using the Merit AAA Server as a base.

The formation of the consortium and the transition from a university based software package to a commercially licensed package brought the realization that current testing procedures were inadequate. In particular, tools were needed to

- Perform regression testing
- Simulate typical load situations
- Provide performance benchmarks

As far as practical, the tools developed would need to meet the following goals:

- satisfy the requirements above
- be simple and convenient to use
- allow for canned and automated scripts

The RADIUS protocol is a simple UDP transaction based client/server protocol. In a typical interaction, an authentication request message is sent from a Network Access Server (NAS) to the RADIUS daemon. The message contains a prospective user's ID and encrypted password. The daemon either performs the authentication function locally, looking the user up in a local file, database or password file, or hands off the request to a remote RADIUS server to be authenticated remotely. In the latter case, while waiting for a reply from the remote server, the initial server may handle other incoming requests. Once authenticated (or rejected), the server returns a reply to the requesting NAS. Subsequently, the NAS may send accounting messages containing billing information on the user's session. These messages must be acknowledged and processed.

The structure of the Merit AAA Server daemon is arranged so that tasks that may take significant time intervals (consulting a database or password file, for example) are forked off to a child process. When the child process completes, it informs the parent of the result and a response is ultimately

formulated as a reply. In the meantime, the parent process may accept additional incoming RADIUS messages, which may result in more child processes and so on.

The problems involved in adequately testing software of this nature are difficult and complex. Interactions between multiple processes may result in subtle errors. In many cases, however, it may be adequate to formulate an authentication request or accounting message, send it to the daemon and observe the reply. This at least serves to verify the basic functionality. More thorough testing requires loading the server with multiple requests until it shows signs of stress [Myers, Deutch].

Merit has a simple tool used for general testing purposes called *radpwtst*. This is a single threaded tool that sends a message and waits for a reply. As such, it was not sufficient for use in load or stress testing, although it did serve to verify functionality.

Expanding the use of radpwtst in conjunction with *expect* [Libes], was one approach considered. This did not meet the goals of simplicity, ease of use and multiple ongoing tests. The DejaGnu system [Savoye] solves some of these problems and offers the added advantage of a standard testing framework, but also suffers from the single threaded test limitation. Consequently, the decision was made to build a test harness facility. Design goals included:

- Ability to handle multiple simultaneous tests

This was a key requirement. The ability to launch multiple simultaneous tests and match replies to their appropriate origin allows for load testing and benchmarking. Since any given test instance may involve multiple interactions with the remote server, each instance must maintain its own context and exist in a separate thread.

- Extensibility

It was considered important that new tests should be added without having to modify the test harness itself. Since it is not possible to imagine all possible tests that may be employed, a mechanism was required that would provide a great deal of flexibility in the test structure.

- Simple and convenient to use

It was envisioned that programmers would write test code to exercise their RADIUS daemon code and to verify that bugs were fixed and stayed fixed across releases. For this to be reasonable, it had to be relatively easy to use and require a minimum understanding of the test harness internals. That is not to say writing a test is easy; it still requires knowledge of the code being tested and the details of the RADIUS protocol.

In addition, a secondary factor involved portability. The initial development environment was a Sun platform, but there there was a concern that the test software could be ported with minimal effort should the need arise.

Tcl was chosen as an implementation vehicle primarily for its extensibility, portability and ability to serve as a glue language to bring together pieces of existing code under a new framework. An initial prototype was constructed to provide functionality similar to the existing *radpwtst* tool. A loadable set of Tcl extensions was written that provided interfaces for initializing the data structure utilized by the underlying *radpwtst* routines as well as sending and receiving RADIUS messages.

The prototype proved surprisingly useful and a number of different tests were written using the facility. Embedding in Tcl resulted in a powerful programmable test facility. As an example, passwords could be corrupted on a pseudorandom basis and the expected reject response could be looked for. It was also a relatively simple matter to check for one of the key features of the Merit AAA Server: the management of simultaneous user sessions. In this case, an authentication request for a user should be rejected if they had reached a configurable session limit, otherwise it should pass. Failure to follow this behavior is a serious error. It was a relatively simple task to write a Tcl test procedure that tracked open sessions and reported when either authentication failed when it should have succeeded or vice versa.

The initial success of the prototype was encouraging, but key features were lacking, particularly the management of simultaneous tests, i.e. multithreading. Although more flexible than the radpwtst tool, it was still single threaded. In addition, writing tests involved, in effect, writing an entire Tcl program. Consideration of these issues led to the current design.

2 The Test Harness Design

The design presented is based on a round-robin test scheduler [Tanenbaum, Say] or executive, interpreting test *templates* [Stocks]. Templates specify test actions which are Tcl procedures. Templates are compiled into an internal format and executed by the test executive.

The test harness design is centered around the test specification or template. The syntax for the test template is shown below in extended BNF:

```
<name>: <test>

<test>::=  <setupProc>
           [<delay>] [<postProc>]
  {
  [ F: <test>]
  [ S: <test>]
  }
<setupProc> ::=  <Tcl procedure name>
<postProc>  ::=  <Tcl procedure name>
<delay> ::= <numeric value>
        | $<Tcl global variable name>
        | <open bracket> <Tcl procedure>
          <close bracket>
<open bracket>  ::= [
<close bracket> ::= ]
```

Test clauses may be nested to any depth providing the ability to develop a decision tree based on whether or not a particular test clause succeeded (S) or failed (F).

The setupProc is a Tcl procedure that sets up the parameters for the next outgoing RADIUS message. On return from the procedure, the test harness sends the message, puts the test instance in a wait list and continues processing other tests. When a response is received that matches the waiting test, the postProc is executed. This is another Tcl procedure that can examine the data returned from the RADIUS server. The postProc decides, based on the returned data, whether the step succeeded, failed or should be repeated.

The syntax describes a test object, which is the basic entity managed by the test executive. When a test is launched, a thread is created with an instance specific data area that continues to exist for the life of the test. This area maintains context and provides private data storage that may be used to communicate information from one step or procedure to another. At the completion of the test, all associated data storage is freed. The test harness allows instantiations of different tests as well as multiple instantiations of the same test to exist independently and simultaneously, each in their own thread. Communication between test threads is handled by global variables and default data values that may be defined for each sequence of tests.

As a concrete example, consider the following test template:

```
 1  session: send_auth auth_recv
 2  {
 3    F: log_bad_auth
 4    S: send_acct acct_recv
 5       {
 6       F: log_bad_acct
 7       S: send_acct_atop \
 8          [global max_session_length; \
 9          randno $max_session_length] \
10           acct_recv
11          {
12          F: log_bad_acct
13          }
14       }
15  }
```

Although shown as block structured, the parser is actually free format and all of the text might have been written on single line.

Line 1 contains the test name (session), the initial setupProc (send_auth) and the initial postProc (auth_recv). When the test is started, the send_auth procedure sets up the parameters for the initial RADIUS message. On return, the test executive sends the message and puts the test instance in a wait list. When an appropriate reply is received, the postProc auth_recv is executed. This procedure looks at the returned information and determines if the test succeeded or failed and returns an appropriate code. On a failure, the test executive evaluates the F branch on line 3 (log_bad_auth) and the test terminates. On a successful return, the send_acct setupProc on line 4 is executed followed by the acct_recv postProc when an appropriate reply message is received. Once again, the postProc decides if the communication exchange succeeded or failed and either the log_bad_acct procedure on line 6 is executed or

the send_acct_stop and acct_recv procedures on lines 7 through 10 are evaluated. In the latter case, the Tcl expression enclosed in brackets is evaluated to generate an integer delay time. This is an optional value that may be used to add time delays in a test. When this value is not present, the test will execute each branch as rapidly as possible. In this instance, a procedure is being called to generate a random interval using a parameter set in a global variable, presumably set in supporting routines. On a successful return from acct_recv, the test terminates since there are no further test clauses.

This design provides a number of advantages. The overall test is described in a compact manner. Elements within a test description are Tcl procedures allowing almost unlimited flexibility and behavior options. At the same time, this allows for libraries of standard procedures to be constructed. New tests can then be constructed from basic building blocks.

Each test procedure is evaluated atomically by the test executive. Consequently, there are no concurrency issues when accessing shared data and variables. Test specific information is carried within an *instance data structure* which exists for the life of the thread and, in effect, provides an instance specific Tcl name space. When the test is started, the data area is created and a handle associated with the area is passed to all test procedures. A library of procedures based on the prototype implementation allows access to the data structure used to create the RADIUS message. Most test procedures consist of a number of calls to either set up the data structure or examine the returned data. The send_auth procedure, for example, might look like this:

```
proc send_auth {ds} {
#-------------------------
# Set up the authentication request
global lastid user auth_port
global authtype rlm

# Get a user/password pair.
# Go round robin on it
incr user
if {$user > $lastid} {
    set user 1
    }

set usernm [format "%d%s" $user \
      [idSuffix $authtype]]

set userpw [makePasswd $usernm]

if {[string length $rlm] > 0} {
    set usernm [format "%s@%s" \
      $usernm $rlm]
    }

setRequestType $ds 1
setUser $ds $usernm
setPassword $ds $userpw
setPort $ds $auth_port
clearAvPairs $ds
setSeqNo $ds [ reserveSN ]
puts "Auth request for $usernm"
return
}
```

As can be seen, this is a standard Tcl procedure. The *ds* parameter is the handle to the instance data structure. The global variables are defined and initialized prior to the test execution. Test IDs used for authentication are generated on the fly in a round robin manner with the password generated from the ID by the *makePasswd* function. The bold text indicate functions which manipulate the RADIUS data structure.

3 Implementation

An overview of the architecture is shown in Figure 1.

A file containing the test template, possibly containing Tcl procedures used in the test, is compiled by a template parser. This is a Finite State Machine based parser which performs syntax checking and compiles the template into an internal tree format. Any Tcl procedures found in the file are added to the Tcl interpreter.

3.1 The Parser

The parser tokenizes the input making heavy use of the *regexp* function. The tokens are consumed by a Finite State Machine module comprising a scant 50 lines of Tcl code. In part, this compactness is due to the FSM description used. This is a Tcl list where each element is a tuple consisting of a state

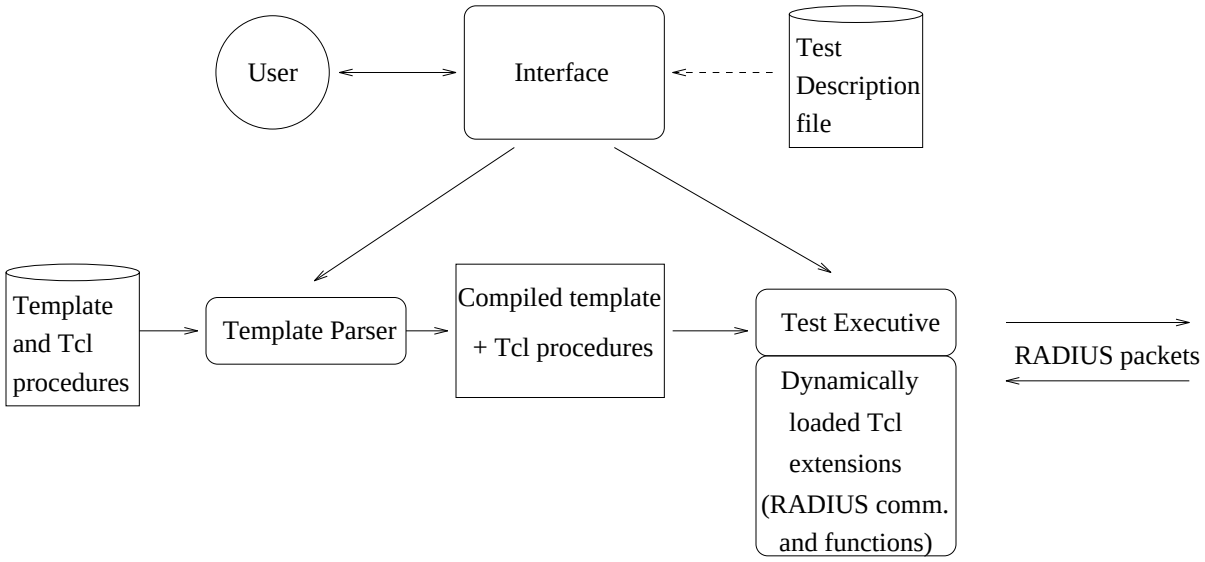

Figure 1: Architecture overview

name followed by one or more token/script/next-state tuples.

As an example:

```
{Start {{? {tclLabel} Start2}}}
{Start2 {
      {L {testLabel getnext} startTest}
      {X {} End}
      {null {} End}
      {? {{parseError "Test must start
          with a label."}} End}
      }
}
```

The state machine starts in the *Start* state and continues until it transitions to the *End* state. In the first rule any token matches the rule and the procedure tclLabel is called. This procedure checks for the reserved label *tcl_code:* which indicates that Tcl procedures following the label should be added to the interpreter. If found, this procedure sets the token type to X and the FSM will exit after it transitions to state Start2.

In state Start2, a label token (L) will cause the two procedures testLabel and getnext to be executed. The testLabel procedure sets up the environment for a new test while getnext causes the next token to be fetched. This rule will then transition to the startTest state. If an end of file condition was encountered, resulting in a null token, the FSM will transition to the End state. Any other token will result in the procedure parseError being called with the error message shown.

The parser uses the *eval* Tcl function to evaluate each procedure in the procedure list. Procedures requiring parameters, such as parseError above, are simply enclosed in curly braces so they become single elements in the list.

The code breakdown for the parser is as follows:

Function	Lines of Tcl
Tokenizing code	190
FSM code	102
State Machine description	61
Routines referenced by FSM description	242
Debugging routines	81

The line count includes comments and white space which inflates the line count 20-50 percent. The FSM code, for example, actually contains only 50 lines of active Tcl code.

Compiled test templates are stored in the Tcl global name space as an array called tst_*testname* with each test step stored as a numbered element of the array starting with array element 0.

3.2 The Test Executive

The test executive is at the heart of the test harness. It provides the basic functionality required to maintain multiple ongoing test threads. The executive is based on a simple Finite State Machine. When a test is started, the associated Tcl setup procedure is evaluated, the resulting RADIUS message is sent and the test goes into a Wait state. At the same time, an entry is place in a timeout list. If a response is not received by a settable timeout interval, the message is retransmitted and the retry count is decremented. If no response is received by the time the retry count reaches zero, or a total timeout interval has been exceeded, the test state will change to either R (retry) or X (exceeded) and the associated postProc will be executed. The values of the timeout interval and retry count may be controlled by the individual test. Following the postProc, the test will either end, if there are no more steps, continue processing with the next setupProc in the test, or delay some time interval before continuing with the next setup procedure. Figure 2 illustrates the FSM used for evaluating the tests.

Tests are loaded into the execution queue after they have been successfully compiled through a call which specifies the name of the test, the number of tests to run and an interval value used to space the tests. For added flexibility, the interval value may be either a fixed time interval, or a procedure which returns an interval value. While the interval may be specified to millisecond resolution, it determines the minimum time that will elapse between tests. The actual time will vary depending on the internal state of the test executive.

The test executive supports a number of additional functions to simplify writing test series. At the start of execution, prior to launching any tests, the test executive will execute the procedure *test_init* if it exists in the Tcl name space. Similarly, after all tests have completed, the procedure *test_end* will be executed. These procedures may be used to initialize global variables, handle logfiles and manage general housekeeping.

For each series of tests, a similar process occurs. In this case, the procedures are called *test_name*_init and *test_name*_end where *test_name* is the template label. These procedures are passed a handle to an instance data structure that will be used to initialize the instance data structures of every test in the series. Default initial conditions may be prepared or other tasks specific to the test series may be performed.

The test executive is written in Tcl with some supporting functions in C. The C functions provide for management of the instance data areas, access to the communications data structure used to formulate the RADIUS message and management of the communications functions, including matching incoming replies to the originating test. The management and execution of the test instance is handled by Tcl functions.

A peculiarity of the RADIUS protocol is that the only piece of information guaranteed to be present in the reply message that may be used to match incoming replies with outgoing requests is an 8 bit sequence number. This allows only 256 outstanding messages from a single source which would be a severe limitation for any type of load test. The current implementation solves this problem by using an array of sockets, any of which may have up to 256 outstanding requests. An array of 20 sockets has proven adequate in general use.

The initial implementation used a polling mechanism to manage the various state lists. Significant performance improvements were obtained by rewriting the C functions to use the dual-ported Tcl8.0 object interface. A final rewrite replaced the polling method with the Tcl event loop using timer (via the *after* function) and *vwait* events. Since the communication functions use C functions, the *fileevent* Tcl command was not applicable and an additional event type was required to indicate when incoming data became available.

The test executive totals approximately 480 lines of active Tcl code, excluding white space and comments. The supporting C code totals approximately 4000 lines. In addition, the test executive leverages the use of 15,000 lines of C from the Merit AAA Server code base to handle many underlying requirements when dealing with the RADIUS protocol.

As a side note, although only the RADIUS communications protocol has been referenced, the

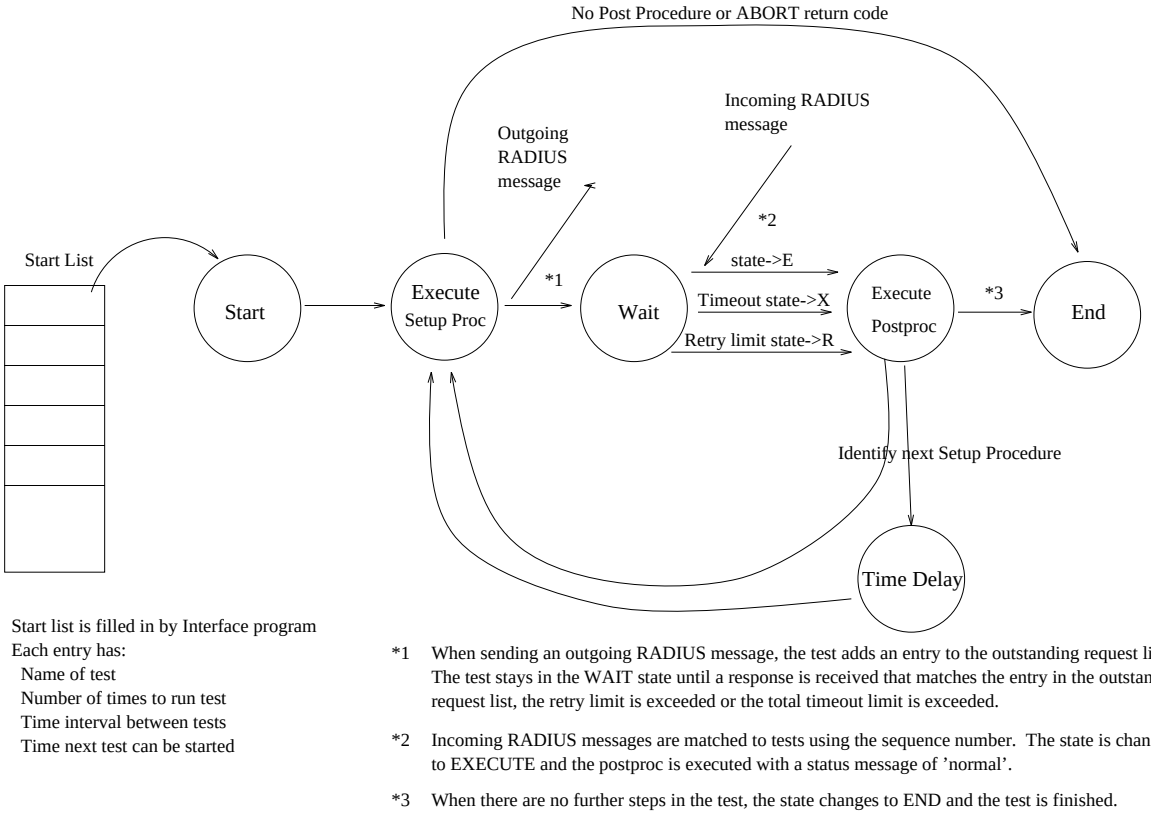

Figure 2: Test Executive State Machine

test executive was written to allow the addition of other communication modules with minimal changes. Test scenarios were envisioned where out of band communication with some aspect of the server environment would be necessary, mandating this ability.

3.3 The User Interface

The template parser and the test executive combine to form a powerful test driver with a great deal of flexibility. The user interface serves to glue these pieces into a usable tool and focus the test onto specific tasks. To date, four different interfaces have been constructed.

- A generic test interface

This interface is a general purpose tool that is used for executing specific template files. The interface supports a number of standard command line arguments as well as a mechanism for passing arbitrary command line strings to the Tcl procedures in the template file.

- A session simulator

This interface simulates remote user login sessions sending authentication and accounting requests. Command line arguments allow session duration and generation rate to be specified. This is used to exercise the Merit AAA Server as well as to perform load testing.

- A file driven session simulator

This interface is similar to the session simulator, but the sessions are defined by a file containing timing and user information and possibly optional parameters for each session. This can be used to repeatedly generate a known user load with a specific makeup.

- A parameter driven test generator

This interface reads a test description file that describes each interaction with the remote RADIUS server. A default mechanism allows a basic RADIUS message to be defined that may be altered and customized by each following test clause. These customization strings are syntax checked and converted into Tcl commands stored in the instance data area. The template Tcl procedures use the eval function to execute these commands customizing the specific test instance. This interface has been used for regression testing and benchmarking.

All interface programs are roughly similar, share many of the same functions and are all approximately 300 lines of Tcl. The parameter driven test interface also includes another 200 lines of Tcl in support routines, primarily for parsing the test description file.

4 Performance

Initial development work for the test harness was done in Tcl7.6 running on a Sun Ultra-1/140. Under this environment, parsing a typical test template file required 1-2 seconds.

In order to better gauge test throughput, a test template was constructed that would immediately fail after sending a RADIUS request. This would cause the post procedure to be scheduled for execution immediately after the setup procedure completed. The setup and post procedures contained only a few Tcl commands necessary to gather statistical information, minimizing the time required to execute these procedures. This template provided an upper limit of the test throughput.

Using the initial polling implementation and Tcl7.6, the development system achieved a sustained throughput of 19 test steps per second. Although adequate for functional testing, this is far below the level required for load testing. Although not stated during the design stage, a goal of 1000 transactions/second was targeted as the utility of the test harness as a loading tool became apparent with use.

Using Tcl-8.0 with no other changes increased the throughput to 26 per second. Rewriting the internal C functions to use the Tcl 8.0 dual ported object interface and modifying some of the internal data structures to decrease search times increased the test throughput to 96 test steps per second.

Finally, rewriting the test executive to employ the Tcl event loop and removal of the polling loops allowed the development system to reach 149 test steps per second. Running this version on a DEC 500MHz Alpha achieved 340 test steps per second. This throughput is sufficient to meet the goal of 1000 steps per second if run as multiple processes on a multi-processor system.

5 Conclusions

Most of the goals set at the start of the project have been reached. The test harness has been in use for new release testing and benchmarking purposes. Canned tests using the defined interfaces are simple and convenient to use for the programming staff. The use of Tcl has allowed for a highly flexible test environment, rapid prototyping and development and the ability to leverage a significant body of existing code into the system. Development of key parts of the system often required only a few days due to the powerful nature of some of the Tcl primitives combined with the interpretive environment. Rewrite of the system from polling to event driven, for example, required only two days.

When running, the test harness typically uses 85-95 percent of the available CPU resources. In CPU intensive applications such as this, attention to execution details is very important. As an example, performance gains of 20 percent were observed when list structures were modified to minimize searching.

Performance is more than adequate for the majority of testing. The goal of 1000 transactions per second appears to be reachable using an appropriate multiprocessor hardware platform. Even the 130-140 transactions per second available in the development environment is sufficient to overload a targeted server on many of the commonly available hardware test platforms in use at Merit.

Porting to the DEC system mentioned in the Performance section was trivial requiring only recompilation on the target system. The test harness has been ported and run on SunOS, Solaris, DEC Unix,

Linux and BSDi with no significant changes.

The most difficult area has proven to be the development of an interface and template combination that balances ease of use and flexibility. Members of the programming staff are not yet confident of their ability to develop new tests at this juncture. This area should improve over time as experience is gained in developing tests to exercise new features added to the software product.

6 Code Availability

Interested parties should contact John Vollbrecht at Merit Network, Inc. (jrv@merit.edu) for licensing details.

References

[Deutch] Michael Deutch, *Software Verification and Validation*, Prentice-Hall, Englewood Cliffs, NJ (1982).

[Libes] Don Libes, *Exploring Expect*, O'Reilly & Associates, Sebastopol, CA (1995).

[Myers] Glenford Myers, *The Art of Software Testing*, John Wiley & Sons, New York (1979).

[Savoye] Rob Savoye, The DejaGnu Testing Framework, `http://darkstar.cygnus.com/rob/dejagnu_toc.html`, (1996).

[Say] Janche Say, Ke-Hsiung Chung, Vernon Rego, *A Simulation Testbed based on Lightweight Processes*, Software Practice and Experience, **24**(5) (May 1994), p. 485-506.

[Stocks] Phil Stocks, David Carrington, *A Framework for Specification Based Testing*, IEEE Transactions on Software Engineering, **22**(11) (November 1996), p. 777-793.

[Tanenbaum] Andrew Tanenbaum, *Operating Systems, Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ (1987).