Using Tcl for Test Automation of a Large Fiber-Optic Switching System

Hugh Dunne

Ciena Corporation
10480 Ridgeview Court
Cupertino CA 95014

hdunne@ciena.com

ABSTRACT

We describe a Tcl-based test automation system for fiber-optic telecommunications equipment.

Keywords

Test Automation, SONET, CORBA, TL1, TclBlend, Expect

1. INTRODUCTION

Ciena Corporation's Core Director™ is an intelligent, high-performance optical networking core switch. It offers state-of-the-art capabilities for real-time provisioning and grooming of fiber-optic networks, supports a wide range of optical interfaces, replaces the functionality of diverse legacy equipment, and dramatically reduces the cost of deploying, operating, and scaling optical networks.

Physically, the Core Director occupies a standard telecommunications equipment bay and can accommodate up to 32 line module cards, each of which can hold up to 8 optical modules. Each optical module has a transmit and a receive port where data is converted between optical and electronic format. A smaller version, the Core Director CI, is roughly half the size and hosts fewer optical modules, but has the same management interfaces:

- CORBA
- TL1
- Proprietary command line interfaces for debugging
- HTTP
- Proprietary XML-like interface
- GUI client written in Java
- NBI (North-Bound Interface) - an IDL-based network management system

When a Core Director powers up, it uses discovery protocols to exchange information with other nodes and form an internal map of the network. This allows sophisticated route optimization and protection policies to be implemented automatically.

2. TESTING THE SYSTEM

Testing this system presents a number of challenges. It must pass a large suite of tests to demonstrate standards compliance. It supports a rich set of interfaces, all of which must be exercised.

In addition, the test automation infrastructure must interface with a large database of test cases, and with various pieces of test equipment which are used to generate traffic to the system under test and to simulate error conditions.

The test infrastructure must also deal with multiple versions of the target system in heterogeneous networks, and even with the possibility that an individual node may have its version upgraded during the course of a test, since it is required to allow upgrading without dropping traffic.

Multiple modes of testing must be supported, including daily sanity tests, regression and stress tests, long-running standards-driven test suites, and informal on-the-fly tests. The test system should be easily used by engineers without significant programming background. It should be highly configurable to deal with an environment in which the availability of physical resources changes rapidly. It should offer a migration path from fully manual to fully automated testing, to overcome user resistance.

3. ADVANTAGES OF TCL

Tcl offers many advantages in this environment:

- It allows rapid prototyping and development.
- It is easy to learn and use.
- It is an ideal glue language and can interface with virtually anything.
- It is powerful and flexible.
- Performance is acceptable in most cases.
- Critical sections can be recoded in a compiled language without having to overhaul the rest of the system.

4. CODE ORGANIZATION

The code for the Test Automation Tool (TAT) is organized hierarchically. At the lowest level are packages supplying procedures for interacting directly with the system under test through its various interfaces. Utility packages provide commonly used procedures, e.g. for creating or parsing common types of Java objects. Another layer of packages correspond to the services running on the node, which correspond to the various connection entities that must be created and configured to provision a network, such as cross-connections, subnetwork connections etc. Packages at this level can be used for simple, on-the-fly tests where a small number of connection entities are set up, used and torn down, and the results are monitored in real time.

For tests which make significant demands on system resources, a further layer of packages allows test suites to be packaged and run in batch mode. These tests are organized according to the major feature that is being tested, such as fault management and the various types of protection schemes. Lower-level procedures are called, the results are compared with expected values, and the test status is recorded. Another layer of packages provides utilities for batch tests, including setup, logging, and a Tk-based interface for point-and-click launching of test suites.

Since many dozens of packages exist, their `pkgIndex.tcl` files have been consolidated into a single file at the top of the hierarchy. This avoids the need for having an unmanageably large `auto_path` which is subject to frequent changes. The single `pkgIndex.tcl` file is auto-generated by a script which traverses the hierarchy and notes the location and version number of every package. Editing of the index file by hand is discouraged since an error can destroy the integrity of the file and prevent many packages from loading.

Coding standards and conventions are important in keeping our large and ever-increasing code base maintainable. Packages follow a standard template, with package and procedure headers documenting the code, explaining procedure usage and including maintenance notes. Each package defines its own namespace and gathers internal variables together in an array which is private to the package. The `namespace export` command is used to indicate which procedures in a package may be called from outside the package, and other procedures should be considered for internal use only, even though Tcl does not enforce this.

5. SYSTEM INTERFACES

5.1 CORBA

CORBA (Common Object Request Broker Architecture) is a standard for distributed computing developed by the Object Management Group. It allows a client to invoke a method transparently on a server object, which may be local or remote. A CORBA module is specified by its IDL (Interface Definition Language), which describes the services offered by the module in a language-neutral manner. The CORBA architecture allows interoperability between applications on different machines in heterogeneous distributed environments.

Each Core Director node has an on-board CORBA server which is the principal management interface for configuring and provisioning the node. For security reasons a CORBA naming service is not provided. Instead, a web server runs on each node and serves the IORs for each CORBA module on a password-protected page.

Although Tcl can interface directly with a CORBA server using the Combat extension [3], we have found it advantageous to use Java and TclBlend [1] as a middle layer. Many of the CORBA methods provided by the server take complex arguments with considerable nesting of data structures. Using TclBlend it is simple to build up complex Java objects and extract particular pieces of data from returned objects. Also, Java's introspection facilities are very useful during prototyping and debugging, and the Tcl code is more understandable when compared with the IDL.

Another reason for using Java is that in order to receive asynchronous alarms and events, we must register a callback object with the CORBA server. We use a simple handler, written in Java, which stores event data in a buffer as it is received. Accessor methods allow the buffer to be queried from Tcl. In this way asynchronous events may be dealt with synchronously. This is important as many tests involve triggering an error condition and verifying that the correct alarm has been observed. Obviously such tests should be able to run in batch mode without human monitoring or intervention.

The Test Tool uses the http and base64 packages to connect to the node's web server, supply an authentication string, and retrieve the IORs for all CORBA modules running on the server. Java methods are then called (via TclBlend) to convert the IOR string to a generic CORBA object and narrow it to a reference to a remote object. The methods of the object can then be invoked just as if it were local.

The Test Tool provides a package for each CORBA module running on the server. The procedures in each such package are basically wrappers for the methods in the corresponding module. The caller invokes the procedures with simple arguments (strings,

lists etc.), the arguments are processed into the required Java data structures, the method is invoked with these arguments, and the significant data is extracted from the return value and returned to the caller in human-readable format.

5.2 North-Bound Interface

The North-Bound Interface is also CORBA-based but is network-oriented rather than node-oriented. It resides on a dedicated server and mediates between customer monitoring systems and the nodes of the network, and also encapsulates the database in which system events are stored. It provides methods for discovering and invoking network services, and aggregates events and alarms from the network nodes. The North-Bound Interface is the preferred interface for monitoring large networks.

5.3 XML Interface

This interface is based on a proprietary, binary-encoded XML format and will eventually replace the CORBA interface on each node. It will allow multiple nodes to be aggregated as a single virtual node.

As with the CORBA interface, we use TclBlend and Java as a middle layer. XML parsing is performed by Java classes and low-level utility packages. At a higher level, packages are provided which correspond to the modules running on the server. Though the same set of services exist as on the CORBA servers, the interface is quite different and thus the internals of the packages are different. However, care has been taken to ensure that the packages present the same interface as earlier versions which interact with CORBA servers, and thus higher-level packages which implement test suites can continue to use these packages without needing modification.

5.4 Command Line Interfaces

The Core Director implements TL1, a standard protocol for interacting with telecommunications equipment. It also implements a number of proprietary interfaces which are not exposed to customers. These interfaces allow debugging and diagnostics, software upgrades, and similar functions.

All of these interfaces can be accessed from a telnet session, and thus can be automated with Expect [2]. Various packages have been written which encapsulate the process of issuing Expect commands and hide the sometimes arcane details of the protocols, providing the user with a simplified interface for exercising TL1 commands and other facilities of the command line interfaces. TL1 provides a very rich set of commands and parameters, and exhaustive testing of the interface is not feasible by manual means. Even automated testing cannot exhaust all combinations of commands and parameters in a reasonable length of time, but it is essential for exercising a sufficiently large subset to give us confidence that the TL1 interface is working correctly.

5.5 GUI Client

Node Manager, a client written in Java, is shipped with the Core Director. We use SilkTest, a proprietary third-party tool, for testing the client. This tool can generate Windows events to simulate user interaction with the client, and perform screen scraping to check the client display. SilkTest is in turn controlled by a Tcl package which can issue commands in batch mode and determine the response of the client.

5.6 HTTP Interface

As mentioned above, each Core Director node has an on-board web server which serves IORs on a password-protected page so

that clients can connect to the CORBA server. The web server can also provide some system information such as the build version running on the server, and diagnostic information such as details of the last assert.

5.7 Abstract Interface

There is significant overlap in functionality between the various interfaces. This allows tests in which we use one interface to exercise a service, and another to check the result. This cross-checking enhances our confidence in our test results, and it is desirable to make such tests easier to develop.

The concept of the abstract interface is that the particular management interface (CORBA, TL1 etc.) used to invoke a given service should be abstracted out, and should simply be another parameter passed to the procedure which wraps the service. Hence the abstract interface has been created as a package which resides one level higher than the packages which wrap particular management interfaces. To invoke a service we specify an interface as the first argument, and the remaining arguments contain the data needed to invoke the service, in a format which is neutral in respect to the interface used. The Abstract Interface package then reformats the data as necessary and calls a procedure in the package which handles the selected interface. The benefits of this scheme are that test suites may be developed more rapidly and greater coverage achieved, with less of a learning curve for the test script writer. For example a simple sanity test may consist of looping over all available interfaces, exercising the same service on each interface with the same arguments.

6. THE MASTER-SLAVE ARCHITECTURE

6.1 Problems with heterogeneous networks

Many test cases require the use of multiple nodes, and a problem arises when the network contains two or more nodes running different versions of the CORBA server. Each version requires a different value for the CLASSPATH environment variable in the Test Tool process, and TclBlend does not allow the CLASSPATH to be changed dynamically. As noted above, the CORBA version running on a node may change during the course of a test. Having to determine the version and perform a switch statement based on the result would rapidly lead to bloated and unmaintainable code.

The solution we have implemented is to share the responsibility among several processes which cooperate in a master-slave architecture and communicate over sockets. The initial Tcl process started by the user is designated the master process. When the user issues a command e.g. to create a connection termination point (CTP) on the node, the master process queries the node to determine the IDL version. It then checks whether a slave process exists to handle that version. If not, it uses the `exec` command to fork another Tcl process, the slave, with a CLASSPATH environment variable set up appropriately for the IDL version. The slave creates a server which can execute commands passed to it from the master. (For security, connection requests which are not from the local host are rejected.) The master then passes the user's command over a socket to the slave instead of passing it directly to the node. The slave issues the command to the node, gets the response, and passes it back over the same socket to the master. Finally the master passes the response to the user.

Note that a single slave process may address different nodes if they are running the same IDL version. The following diagram shows a schematic view of the setup.

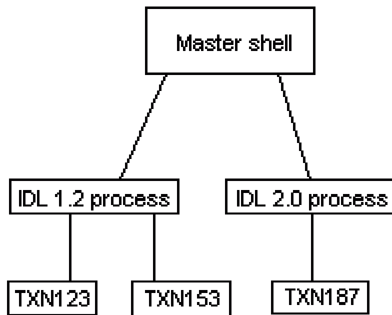

Figure 1. The master-slave architecture.

This entire process is transparent to the user; the package interface is the same as when a single process is used. The only difference is that in the single-process case, the user must know which IDL version is running on the target node and select the corresponding version of the test tool package (by using the command e.g. `package require -exact CTP 2.3`). If no package version number is specified, the highest-numbered version is loaded; this is the version which implements the master-slave scheme.

6.2 Expect

Another use of the master-slave architecture is in overcoming limitations of Expect support on Windows. Expect on Windows NT is only supported for Tcl 8.0 while TclBlend is one of several factors forcing us to use later versions of Tcl. To solve this problem we provide a wrapper package around Expect and prohibit other packages from using Expect directly. This wrapper package functions as the master. It forks a single slave process which runs Tcl 8.0 and creates a server as described in the previous section. The master process passes Expect commands to the slave, which executes them and returns the result to the master.

6.3 North-Bound Interface

The master-slave architecture is applicable to a variety of situations where a solution involving a single process would not be powerful enough or would lead to excessively complex and unmaintainable code. By using cooperating master and slave processes we can take a building-block approach to extending the functionality of the system and rapidly deploy powerful tools without having to maintain a large monolithic program.

A limitation of TclBlend is that it can hang the Tcl session if a large number of java objects is created in a short time. This is a problem with the NBI since events are aggregated from several nodes, and this large numbers of events may be received by the client.

To overcome this problem, a modified master-slave approach is used. A master Tcl process controls a system of cooperating slave processes, including a Java process which interfaces directly with the NBI server. The Java slave may easily be made multi-threaded using standard Java classes. Each thread communicates with a slave process, written in pure Tcl, which provides parsing and filtering services using Tcl's powerful regular expression features, and also simplifies the collection of performance statistics using Tcl's `time` command. By running the Java Virtual Machine in its own process we avoid problems with TclBlend being overwhelmed by too many objects. This sharing of responsibility uses the strengths of Tcl and Java while overcoming some of the weaknesses of each language.

7. CONCLUSION

The Test Automation system has evolved from a small prototype into a large suite of libraries and scripts totaling over 100,000 lines of code, developed by several programmers in different geographic locations. Its user base has also grown. Originally a tool for supporting the test and validation team, it has come to be indispensable for meeting testing deadlines, for enabling developers to get immediate feedback on new functionality, for running basic smoke and sanity tests on new builds before submitting them to full test suites, for provisioning fully-loaded nodes and large networks (which would be a very lengthy and tedious process using a GUI) and as an aid to tech support personnel in the field.

The main lessons learned from this project are as follows.

Close liaison with development is essential. In a dynamic environment where new versions of the system under test may force an overhaul of the test automation infrastructure, the automation team must stay in the loop and be proactive regarding changes to the system under test that will have a major impact on how testing is done. Development engineers are often unfamiliar with Tcl, and unaware for example that it is an interpreted rather than a compiled language. One consequence of this fact is that mismatches between the test tool and the target will not be caught at compile time (since of course there is no compile time) but may manifest themselves in unpredictable ways at run time. Good communication between developers and the automation team, and an understanding of the dynamic nature of Tcl, are very important for tracking down the cause of unexpected behavior.

User education is also essential and must be an ongoing process. The initial version of the Test Tool required a fair amount of manual setup by the user. The process of installing and setting up the tool has been considerably automated and simplified but this does not eliminate the need to document the tool. Detailed instructions for installing the tool, lists of common problems and their solutions, and tools for browsing the available test libraries have all proven useful in familiarizing users with the tool. This familiarity is reinforced by periodically emailing the user community with troubleshooting checklists and pointers to the various resources available.

Finally, backward compatibility is a virtue, but good judgment must be shown when the point of diminishing returns is reached. As the system under test has developed, we have taken care to ensure that previously developed test scripts and packages remain useful in spite of changes, but when changes in the underlying system accumulate past a certain point, a clean break with the previous version of the tool is less painful in the long run than continuing to try to accommodate changes incrementally. Fortunately Tcl's power and flexibility, in conjunction with its rapid prototyping strengths, support both types of change, in effect giving us the best of both worlds.

8. ACKNOWLEDGMENTS

The author thanks Ciena Corporation and in particular Chris Cook and Richard Genet for their support in preparing this paper.

9. ABBREVIATIONS

CORBA	Common Object Request Broker Architecture
CTP	Connection Termination Point
HTTP	HyperText Transport Protocol
IDL	Interface Definition Language

IOR	Inter-ORB Representation
NBI	North-Bound Interface
ORB	Object Request Broker
TAT	Test Automation Tool

10. REFERENCES

[1] Dejong, Mo and Hylands, Christopher. TclBlend. http://www.tcl.tk/software/java/

[2] Libes, Don. Exploring Expect. O'Reilly & Associates, 1994.

[3] Pilhofer, Frank. Combat. http://www.fpx.de/Combat.