Remote Control of Test Equipment Using Distributed Objects

Timothy L. Tomkinson
Fellow Software Engineer
Northrop Grumman Electronic Systems
Baltimore, MD

15th Annual Tcl/Tk Conference
October 2008

Abstract

Many testing environments consist of a heterogeneous mixture of CPUs connected via a network. These CPUs, in-turn, are used to control various pieces of test equipment over a wide variety of interfaces. By combining the power of Tcl, the object-oriented capabilities of Itcl, the remote communication facility of comm and the ability of Ffidl to access C libraries, an extremely simple remote method call mechanism was developed that both hides the complexity of network programming and eliminates the need to write control software in any language other than pure Tcl.

Summary

As test systems become more and more complex, it often becomes necessary to distribute the workload among several CPUs on the network. Doing so usually complicates the design by introducing client/server applications and implementing custom messaging protocols. Other complexities involve bridging different operating systems (Windows/Mac/Unix/VxWorks) and different machine architectures (big-endian/little-endian). The control of the test equipment itself usually involves interfacing with various drivers that communicate with their devices using a variety of bus protocols, such as Ethernet, RS-232, USB, GPIB, 1553, VME and PCI. These drivers are usually written in C.

To handle the message-passing middleware layer of the architecture, one could use an off-the-shelf solution, such as CORBA (Common Object Request Broker Architecture) or DCOM (Distributed Component Object Model). However, these packages are usually overly complex, involve a very steep learning curve for untrained developers, are platform-specific, or don't provide a clean Tcl interface. CORBA, for example, requires that the remote method call interface be defined in a C++-like IDL (Interface Definition Language) file. The IDL is run through a compiler that produces an object for the client, known as a stub, and a skeleton object for the server, known as a servant. The skeleton object must then be extended to implement the servant's desired functionality. The client and servant objects are ultimately managed by the CORBA ORB (Object Request Broker), which dispatches client messages to the appropriate servant.

Needless to say, CORBA is exceedingly complicated for most testing environments, however its paradigm is very appropriate. A client application wants to control something remotely. It creates a local object that transparently connects to a remote object somewhere on the network. Methods that are invoked on the local object are automatically invoked on the remote object. Return values from the remote method call are returned to the local caller, including error exceptions. Tcl makes this concept incredibly simple to implement.

An Itcl class/package called Remote was created to handle the remote method call interface. It uses the comm package to handle the underlying network communication. On the server side, a simple call to the `config` procedure starts the server:

```
package require Remote
Remote::config –port 5000 –local 0
```

On the client side, a Remote object is created that specifies the desired server and port number:

```
package require Remote
set remote [Remote #auto $hostname 5000]
```

The Remote object's `send` method is used to send Tcl commands to the remote interpreter:

```
$remote send "package require PowerSupply"
```

This simply passes the command to comm's `send` method. In this example, assume that the PowerSupply package defines the PowerSupply class. The Remote object's `new` method then creates an object in the remote interpreter and returns an object in the local interpreter that is linked to it:

```
set powerSupply [$remote new PowerSupply]
```

The `new` method uses a remote procedure call to create a new PowerSupply object in the remote interpreter. It then uses Tcl's introspection capabilities to query the remote object and determine its public methods and procedures. It creates a local object with the same methods and procedures, but replaces the bodies with remote procedure calls to the remote object. The client can then simply call methods on the local object which will be invoked on the remote object. Return values (and errors) work as expected:

```
$powerSupply setVoltage 5.0
set voltage [$powerSupply getVoltage]
```

When the local object is deleted or the interpreter terminates, the remote object is automatically deleted as well.

To control the test equipment itself, the Ffidl package was used to create Tcl wrappers around the device driver C libraries. For example, the NI-VISA library from National Instruments can control serial, GPIB and VISA-compatible Ethernet devices. A wrapper class called NiVisa was developed to provide a Tcl interface to this library. The power supply driver, for example, can then be written in pure Tcl:

```
package require NiVisa
class PowerSupply {
    public method setVoltage {volts} {
        $niVisa write "VOLT $volts"
    }
    public method getVoltage {} {
        return [$niVisa query "VOLT?"]
    }
}
```

By using a table look-up mechanism of method name versus command string, a device driver can be created with remarkably little code. A driver for a new device can be created in hours instead of days.

Conclusion

Using Tcl and the concept of remote method calls has dramatically increased our productivity when developing distributed test systems. An engineer can develop and test new code on his own PC, then with a few additional lines of Tcl can run his code remotely without modifying it or knowing anything about networking software or the underlying architecture. By using Ffidl, device drivers can be written in pure Tcl. Combining these features has allowed us to create a simple, yet extremely powerful distributed test system that is both easy to use and maintain.