Real-Time Code Distribution on a Heterogeneous Network for Laboratory Equipment Control

Robert W Techentin, techentin.robert@mayo.edu
David R Holmes, III, holmes.david3@mayo.edu
Barry K Gilbert, gilbert.barry@mayo.edu
and Erik S Daniel daniel.erik@mayo.edu

Special Purpose Processor Development Group (SPPDG)
Mayo Clinic and Foundation
200 First Street SW
Rochester, MN 55905

Abstract

A custom distributed computing framework was developed to support electronics research laboratory automation, with the primary goal of flexibility in a heterogeneous network. The custom implementation is based on Tcl [1] and the "comm" package[2], is more flexible than traditional remote procedure call (RPC) methods[3,4] because it is based on real-time distribution of code. This technique could be applied to a variety of network and distributed computing applications.

Introduction

This paper describes an approach to distributed computing that we call real-time code distribution, in which compute servers are both highly flexible, and virtually transparent to the application program. Like many distributed computing techniques (see [5] for review or [6] for a description of Tcl-DP), the remote procedure call [7,8] is transparent to the application program. The application chooses local or remote execution at startup, and the rest of the application code is unchanged. This new approach, however, distributes executable code to the remote server at run time, eliminating the need to install, revise or maintain services on the server system(s). This dramatically increases flexibility for the application program, as it can distribute virtually any procedure or library to the remote server, without incurring management overhead of remote services or the possibility of version incompatibilities between the application and any given server.

Real-time code distribution was developed for the Special Purpose Processor Development Group (SPPDG), at Mayo Clinic. Like many other electronics research laboratories, we increasingly rely on computer controlled electronics test equipment. Any given experiment usually requires several items of test equipment, ranging in complexity from simple servo motors or power supplies to sophisticated sampling oscilloscopes and network analyzers. In the past, experiments were controlled by a single workstation connected to the test equipment by an IEEE 488 (GPIB) bus. A single workstation application would control all the pieces of equipment and send the test data to a centralized file server. However, as modern test equipment has become more computerized and more heterogeneous, instrument control has evolved into more of a distributed network application [9], where each piece of equipment is controllable by some form of computer, sometimes dedicated to that instrument, and sometimes even embedded within the instrument itself.

Unlike many other laboratories, we must constantly rearrange test equipment to perform new experiments, on a weekly or even a daily basis. Because many of the experiments are virtually unique, we require a great deal of flexibility in deploying instrument control applications. However, deploying new software to a constantly changing network of PCs, workstations, and embedded computers is challenging, even in a modern networked environment.

To address our need for flexibility in the test laboratory, we have implemented a client/server network control system that allows an application program to control many different pieces of test equipment connected to many different computers. The application client contains all the code necessary to perform the experiment. Each piece of networked equipment runs a simple, generic server daemon (or service under Microsoft Windows) which can process arbitrary Tcl code. The application client computer sends scripts and commands to the servers, and coordinates their activities by simple procedure calls.

This paradigm has several advantages in the test laboratory. First, the server nodes are very flexible. Client control programs can send literally any code to the slaves, including commands, simple scripts, procedure definitions, and even entire Tcl packages. Once the daemon is installed, it is not necessary to install or maintain any applications on the servers. Second, because the libraries and code are pure Tcl, this remote execution model works equally well with Unix workstations or PCs running Windows. The daemon even works on the dedicated PCs supplied with some test equipment.

The client and server code was easily implemented in modern Tcl technology. Communications is accomplished via the Tcllib “comm” package. The slave installation can be either a “batteries included” Tcl/Tk distribution, or just a tclkit [10] with the “comm” package. The client application can supply custom procedures or even entire Tcl packages through the network interface.

Laboratory Requirements

The impetus for designing a remote computing architecture was an experiment which required three computers: the client system ran the main application which interacted with the operator via a GUI; and two server systems which controlled several servo motors and external sensors. While each external device had a serial/socket interface, custom programming was necessary to develop the Tcl interface for the client application.

Initial application development was done on a test system with direct connections to the server motors and sensors. But since it was known that the actual laboratory deployment would require the client/server architecture, it was immediately obvious that parts of the equipment control code would need to be developed twice: once for “local mode,” and once for the client/server deployment

A related constraint was imposed by the nature of experiments conducted in the SPPDG laboratory. There are several test bays, each of which may be configured with a different experiment. Depending on the laboratory work load, different makes and models of the same type of equipment may be available for a specific experiment. And for multi-day experiments, it is possible that equipment availability could change on any given day. This implies that it might be necessary to update the equipment control packages frequently, and that those updates would need to be deployed to a number of laboratory workstations.

To avoid these possible development problems, the authors chose to introduce transparent remote communications with real-time code distribution directly into the package development cycle. There were three notable requirements which guided the design of the remote communication:

(1) It should appear essentially transparent to the application programmer
(2) It should work exactly the same remotely as it does locally (given the same environment)
(3) It should have very little overhead both on the local side and the remote side

Implementation

There are several ways to implement distributed computing and remote computation within Tcl. Several examples are presented, in order of complexity and automation, to illustrate the mechanism behind real-time code distribution. These examples are presented from the client application perspective, without regard to well known robust server strategies for multiple connections, client authentication, code security (safe interpreters [11]), or service discovery.

One well known networked client/server strategy is based on Tcl socket support, and is described and documented in text books [12] and in wiki examples [13]. The strategy is illustrated in Figure 1 in a UML sequence diagram. The objects named in the rectangles represent the client and server processes, and the descending lines represent time. The arrows between the objects represent procedure calls within or between the processes. The computation server in Figure 1 has pre-defined procedures (e.g., foo) which can be executed on behalf of the clients. It also includes code to recognize network connections, execute the procedures, and return the results to the client.

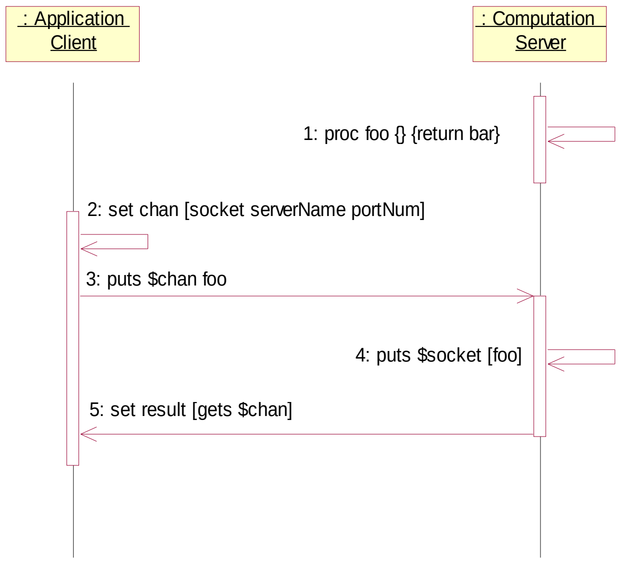
: Application Client
: Computation Server
1: proc foo {} {return bar}
2: set chan [socket serverName portNum]
3: puts $chan foo
4: puts $socket [foo]
5: set result [gets $chan]

Figure 1: Remote execution via socket server (20176)

Figure 2 shows the same functionality implemented with the tcllib "comm" package. Using the "comm" package, with semantics based on the Tk send command, the client simply sends a command to the server, which executes it and returns the result, greatly simplifying both the client and server code. This approach still depends on pre-defined procedures, but using the "comm" package can be quite flexible, as the client can ask the server to execute arbitrary script code. But pushing complex scripts through a network connection could be cumbersome.

A more flexible approach, illustrated in Figure 3, exploits Tcl's dynamic nature, sending procedure definitions through the communications channel. This allows the client to define arbitrary functions, take advantage of byte-compiling on the server, and re-use application code with simple calls to the server. But in this case, it is still necessary to "send" commands to the server. Application code must be written specifically to utilize remote processing.

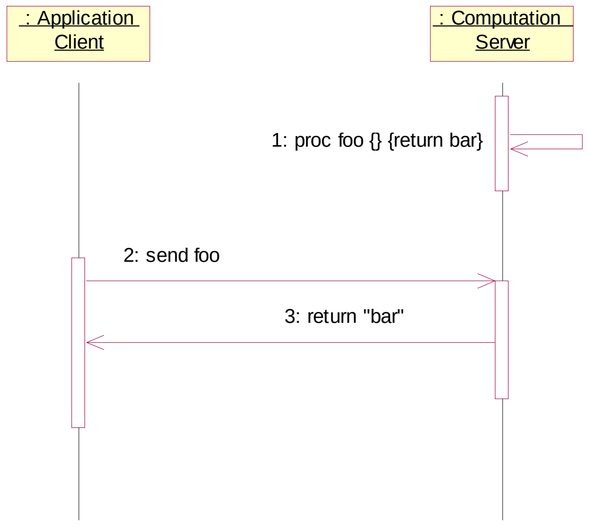

Figure 2: Remote execution via tcllib 'comm' package (20177)

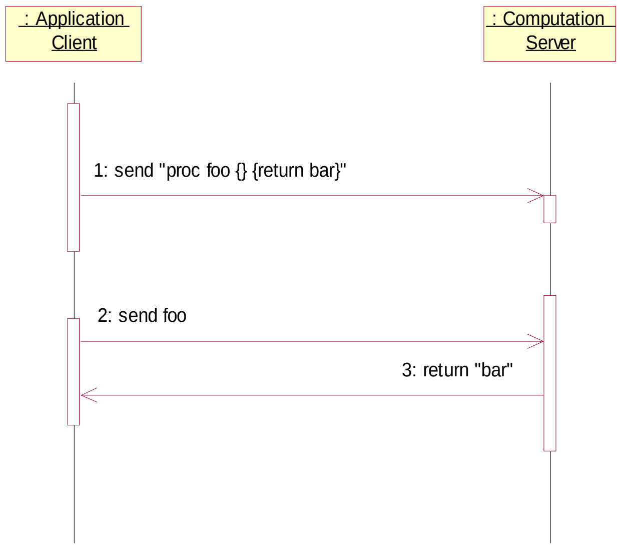

Figure 3: Remote execution of client defined procedure (20178)

Figure 4 depicts an enhancement where the client sends a procedure to the server, then defines a local alias or proxy procedure. This alias procedure simply sends the appropriate commands and arguments to the server. The comm package handles the networking, and the result is returned as if the procedure were executing locally. The comm package also handles errors elegantly, returning them through the alias procedure. Error messages are identical to those produced by local calls, although the stack traces are somewhat more complex. As far as the client application is concerned, the remote procedures are identical to locally executed code. It is no longer necessary to customize the application code to perform remote processing.

The next step in simplifying this approach for the application program is to wrap the process of creating remote procedures and the corresponding aliases into a simple procedure call. The "remoteCommands" procedure shown in Figure 5 will make new commands available to the client interpreter. It accepts an arbitrary script which is sent to the server's "defineCommands" procedure, where it is evaluated to define new commands. The server exploits Tcl's introspection capability to compare available commands and namespaces before and after executing the script. It inspects both the global namespace and any newly created namespaces for new commands, and returns the list of new commands to the client. The client then creates alias procedures to call the remote procedures.

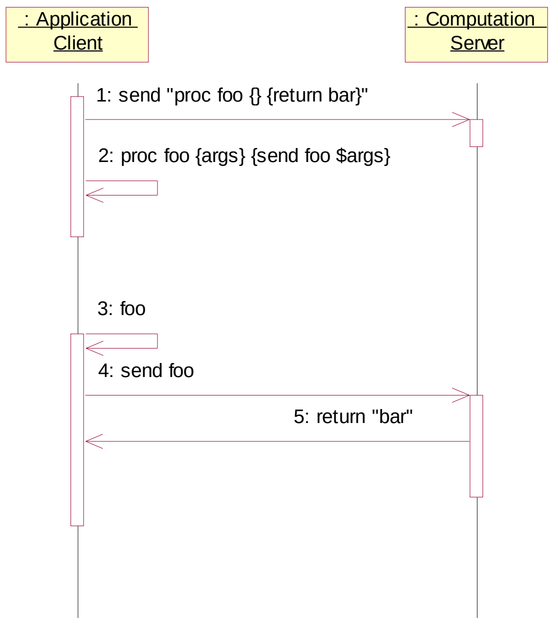

Figure 4: Remote execution of client defined procedure with alias (20179)

The client code might look like this example, where the server is named "mynock", and is listening on port 3456. Note that the client procedure is actually an alias that calls the server.

```
  % remoteCommands mynock 3456
"proc foo {} {return bar}"
  % foo
  bar
  % info body foo
  eval ::comm::comm send {{3456
mynock}} foo {$args}
```

The next level of automation is illustrated in Figure 6, with new code on both the client and server sides allowing the server to "source" client code files. The server renames the core "source" command for the duration of evaluating the client's command script. This new "source" command will request the contents of the file from the client, and evaluate the results. Tcl's introspection capability is essential for this level of integration, as neither the client nor the server may have a complete list of newly define procedures. But as in Figure 5, a list of new server commands are returned to the client, which sets up the aliases. This implementation allows a compact server installation, requiring only a basic Tcl or tclkit installation and the tcllib "comm" package. But the server is not limited to pre-installed services. It can load source code from any file visible on the client system.

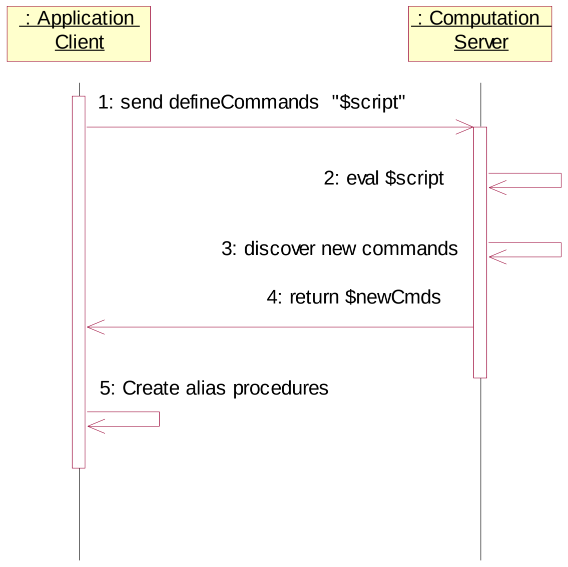

Figure 5: Automatic definition of remote procedures and aliases (20180)

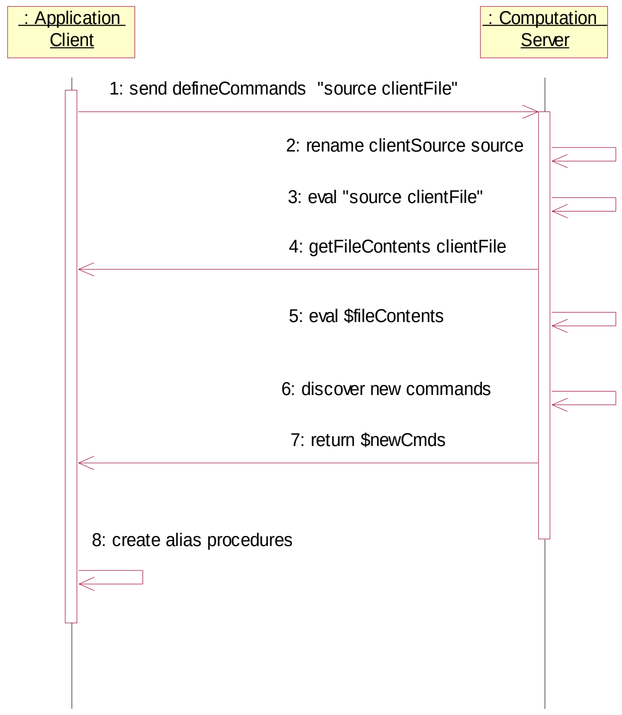
: Application Client
: Computation Server
1: send defineCommands "source clientFile"
2: rename clientSource source
3: eval "source clientFile"
4: getFileContents clientFile
5: eval $fileContents
6: discover new commands
7: return $newCmds
8: create alias procedures

Figure 6: Server loading client source files (20181)

One key feature of the "comm" package is that we can send commands in synchronous mode, so that client execution waits until the remoteCommands definition is complete before proceeding. But while the client is waiting for the server, it can still process events and respond to the server's request for the source code file. The channels do not block, and no additional client side code is necessary to answer the compute server's requests.

The final step in this paradigm is presented in Figure 7, where the server can load entire Tcl packages from the client. The client and server are extended with a pair of routines that hook into the server's "package unknown" facility. When the client sends the server a "package require" command to load an unknown package, the server will query the client for the "if needed" script. For a pure Tcl package, the "if needed" script usually includes a number of source commands, which are already supported by the server and client. The server will load the client's copy of the package automatically.

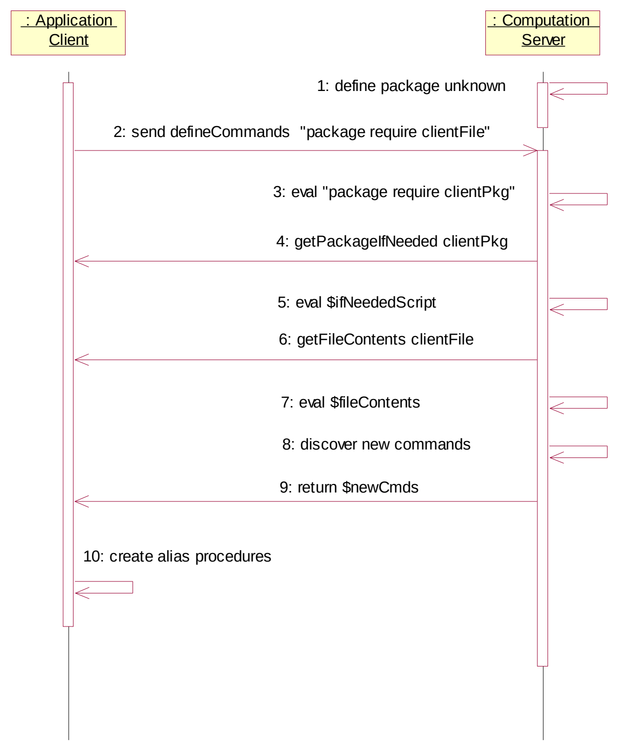
: Application Client
: Computation Server
1: define package unknown
2: send defineCommands "package require clientFile"
3: eval "package require clientPkg"
4: getPackageIfNeeded clientPkg
5: eval $ifNeededScript
6: getFileContents clientFile
7: eval $fileContents
8: discover new commands
9: return $newCmds
10: create alias procedures

Figure 7: Server loading client packages

As an example, a client could request a package like the tcllib "counter" be loaded on the remote server. Since the server does not have the package, it is loaded from the client in real time. All future commands would execute on that server, but they will appear to work as local commands on the client.

```
  % remoteCommands mynock 3456
"package require counter"
  % counter::init x
  % counter::count x
  % counter::count x
  % counter::count x
  % counter::get x
  3
```

It is almost embarrassing that this kind of flexibility can be implemented in three simple procedures on each of the client and server sides. One pair of procedures supports creation of remote commands. A second pair of procedures allows the server to load source code from the client. A third pair of procedures allows the server to request packages from the client.

Implementation in the SPPDG Laboratory

Real-time code distribution has been implemented in the SPPDG test laboratory with satisfactory results. Instead of "compute servers", the "instrument servers" are connected to test equipment, and loaded with a current version of ActiveTcl and extensions for controlling the test equipment. A simple server is installed as either a daemon or a service, and can be accessed from application clients which need to control test equipment.

Experiments are conducted by client applications which dynamically distribute code to the instrument servers. The servers can control the experiment, collect and summarize data, and return the results to the client application. When the experiment is complete, the server is reset, and is ready to accept commands from a new application.

The first implementation of real-time code distribution was designed into the instrument control package. The package would normally operate in local mode, but calling the RemoteConnect procedure would open a communications channel to a server, send the package's source file, and set up alias procedures for each exported command. The server would simply execute commands for the client until it received a "reset" command. The code worked very well, but had the disadvantage of requiring modifications to each package that was to be executed remotely.

A revised implementation takes advantage of Tcl's dynamic nature and introspection capabilities to allow remote distribution of any pure Tcl package. This enhancement required support code on both the client and server sides, but did not require any package customization. Alias procedures are defined for any new commands "discovered" by the server. This has the advantage of being very flexible, but it is more difficult to switch between local and client/server modes during application execution.

Real-time code distribution was found to meet the previously stated design goals. Because the control code is distributed from the client application in real-time, there is no need to update and release libraries for the instrument servers. Creating alias procedures on the client makes the remote code behave as if it is running locally, so there is minimal impact on the client code. And because the code is distributed only when a package is loaded, there is minimal impact to the long running client and server processes.

Future Enhancements

While we have been controlling electronics laboratory equipment, this approach should work equally well for distributing generic computations across a heterogeneous network. Safe interpreters, encryption and digital signatures could be employed to provide application safety. There is significant ongoing research and development into distributed computing “grids” (ref globus project at http://www.globus.org/) that address many of these higher level concerns. Adding real-time code distribution into such a framework would greatly enhance the flexibility of any grid computing installation.

This paper has focused specifically on distributing portable Tcl code in a heterogeneous networked environment. But it is desirable to consider extending this technique to real-time distribution of compiled code. Distributed computing is often compute-intensive, and instrument control often depends on compiled Tcl extensions. It is reasonable to envision real-time distribution of shared libraries by a load-from-client mechanism, similar to the script support presented here. A simple implementation would lock the client and server into the same computer architecture and operating system, although a more sophisticated implementation might be able to support heterogeneous networks. Another approach might be the application of the CriTcl [14] extension to dynamically compile and load C code distributed from the client to the server. This would imply servers that can compile code, and may introduce further security risks, but the potential benefits could be great.

Conclusion

Real-time distribution of Tcl source code was implemented and demonstrated in the SPPDG laboratory. This approach had three distinct advantages. First, it was not necessary to deploy instrument control libraries to the instrument control computers, as the code was distributed from the client application at run time. The only deployment was creation of a very lightweight server. Second, the implementation is virtually transparent to the application programmer: the application code works the same in both local and distributed modes. And third, there is very little overhead in real-time distribution of code.

Real-time distribution and execution of code highlights the power of Tcl’s dynamic nature and networking support. Instead of merely simplifying code deployment with packages, repositories or Starkits, this approach demonstrates the complete elimination of instrument control application deployment in an industrial research laboratory environment.

References

[1] Ousterhout, J. “Tcl: An Embeddable Command Language.” Proc. USENIX Winter Conference, January 1990.

[2] LoVerso, J. “comm - a replacement for send.” <http://www.schooner.com/~loverso/tcl-tk/ >, May 1999.

[3] Winer D. “XML-RPC Specification.” < http://www.xmlrpc.com/spec> , Jun 2003

[4] W3C Proposed Recommendation "SOAP Version 1.2 Part 0: Primer", <http://www.w3.org/TR/2003/REC-soap12-part0-20030624>, June 2003.

[5] Zomaya A. Parallel and Distributed Computing Handbook. McGraw-Hill, New York, 1995.

[6] Smith, B.C., Lawrence R., &Stephen Y. “Tcl Distributed Programming.” In Proceedings of the 1st Tcl/Tk Workshop. June 1993.

[7] Birrell, A.D. & Nelson, B.J. "Implementing Remote Procedure Calls." ACM Transactions on Computer Systems 2, 1 (February 1984): 39-59.

[8] Sun Microsystems. Network Programming Guide, March 1990.

[9] The Proceedings of Workshop on Automated Control of Distributed Instrumentation, April, 1999.

[10] Equi4Software. “Tclkit” <http://www.equi4.com/tclkit.html> , Mar 2004.

[11] Safe-Tcl <http://www.tcl.tk/software/plugin/safetcl.html> April, 2001.

[12] Ball S. Web Tcl Complete McGraw-Hill, April 1999.

[13] Tannenbaum, A. “client/server with fileevent” <http://wiki.tcl.tk/1757>, Sept 2004.

[14] Equi4Software. “Critcl” <http://www.equi4.com/critcl.html> , Feb 2004.