A Zytek Communications Corporation White Paper

Embedded System Daemon Communication Using Tcl And Sockets

Stephen Melvin, Ph.D.

March 15, 2012

Copyright © 2012, Zytek Communications Corporation

Embedded System Daemon Communication

Using Tcl And Sockets

Abstract

A simple and flexible approach to daemon to daemon and daemon to application inter-process communication is presented. Tcl-based daemons communicate with external devices, with each other and with applications over TCP/IP sockets. By passing Tcl command strings in line oriented exchanges, a simple mechanism has been developed that is flexible enough to allow full control of the remote process and is robust and easy to develop and test. The system described has been employed in an embedded system running on a UNIX-based single board computer.

Introduction

Tcl (Tool Command Language) is a scripting language developed in 1988 and has been utilized in a wide variety of applications, from GUIs to embedded systems [1]. The ability to pass command strings between Tcl applications has been widely utilized as a form of inter-process communication [2]. Examples of packages that support such features include the "send" command of Tk [3] and the "comm" package [4]. This white paper illustrates an application of inter-process communication in an embedded system utilizing these basic concepts.

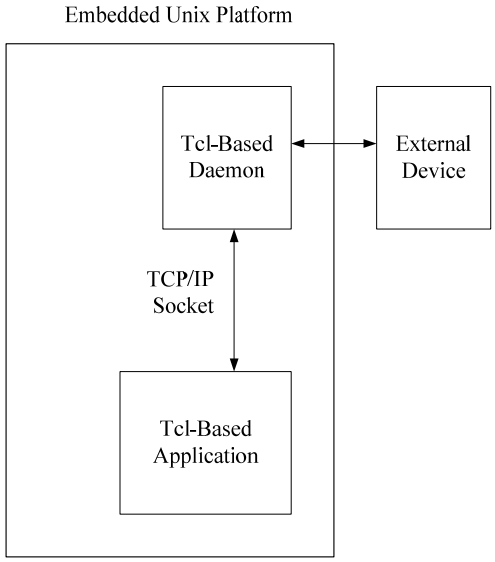

Figure 1

Figure 1 illustrates the basic environment of the inter-process communication mechanism presented in this paper. The basic framework includes a Tcl based daemon that communicates with a hardware device. The Tcl based daemon is started automatically at boot time and runs in the background independently from any user interface. The daemon communicates according to a device specific protocol with a hardware device and listens for new connections on a well known port number. The daemon may also establish its own connections to communicate with other daemons.

The application described in this white paper is based on a simple but effective mechanism for processing input from an application. First a socket is established that waits for a connection on a well known port. When a connection is established the connection is configured for line oriented input and when the socket becomes readable a line of input is retrieved. This line is assumed to be a Tcl command that can be executed with the "eval" command. Such a command passed to the daemon can be used to read or write any variable in the Tcl daemon. The application, which is also a Tcl program, passes the Tcl commands and processes return values in a Tcl context.

In this paper we fill first detail the configuration on the daemon side followed by a description of the application side. Finally, a real-world example of the use of this system is outlined.

Daemon Configuration

First we will discuss how the Tcl based daemon is configured. The Tcl daemon is an ordinary Tcl program that is initiated at boot time and runs in the background. The initiation at boot time can involve simply a line in a startup script. For example, FreeBSD and NetBSD provide a convenient startup system, "rc.d," that allows scripts to be automatically started based on configuration strings in /etc/rc.conf. Alternatively a startup script may simply be placed in /usr/local/etc/rc.d, which will be run automatically at boot time. For example:

```
#!/bin/csh
/usr/local/tcllib/tcldaemon.tcl &
```

The Tcl daemon has five main components: 1. the socket declaration, 2. the connection procedure, 3. the input processing procedure, 4. the polling loop and 5. the command processing procedure. Appendix A illustrates sample code for each of these components of the Tcl based daemon.

1. Socket Declaration

The socket declaration establishes a well known port to listen to for incoming connections. Because it is an ordinary socket connection, multiple incoming connections are supported and local as well as remote connections can be utilized. In the sample code illustrated in Appendix A the socket declaration consists of the following line:

```
socket -server tcld_server 2742
```

In this example a specific port is utilized and the Tcl daemon will listen for all incoming connections on that port.

2. Connection Procedure

The connection procedure ("tcld_server" in Appendix A) is the procedure that is invoked when a new incoming connection on the port is received. This procedure consists of the following command to set the connection to line mode:

```
fconfigure $channel -buffering line
```

and the following command to define which procedure to utilize ("tcld_read" in this example) each time the socket becomes readable:

```
fileevent $channel readable [list tcld_read
$channel]
```

The connection procedure can also include diagnostic and/or logging functionality if desired. This procedure will be invoked once for each incoming connection that is established.

3. Input Processing Procedure

The input processing procedure ("tcld_read" in Appendix A) is the procedure that is invoked each time a line of input is received on the incoming connection. Each received line is assumed to consist of a single line Tcl command string, which is evaluated using the Tcl "eval" command. If a return value is expected, the command string should set a local variable named "retval". If "retval" is not set by the command "OK" is returned upon successful completion of the command and "ERR" otherwise.

The main components of the input processing procedure are the command to retrieve the line of input:

```
catch {gets $channel line}
```

the command to execute the Tcl command received:

```
catch {eval $line}
```

and the command to return the result to the remote application:

```
puts $channel $retval
```

Thus, this mechanism allows both input and output oriented commands. For reading information from the Tcl daemon, commands are sent that return a list of variables in the special "retval" variable. For writing information to the Tcl daemon, commands are utilizes that set other variables. These sequences are illustrated in the Application Configuration section below.

Note that because the input processing procedure is a local function with the daemon script, every variable that a remote process needs to read or write must be declared global in this procedure.

4. Polling Loop

The polling loop ("poll" in Appendix A) runs independently of any input and updates variables associated with the daemon activity. In the example illustrated in Appendix A, two global variables "pos_x" and "pos_y" are updated by the polling loop. The polling loop is initiated the first time by the main procedure and then re-invokes itself using the command:

```
after 1000 "poll"
```

which causes it to repeat at the specified interval.

5. Command Processing Procedure

The command processing procedure, which is the main loop in Appendix A, executes code stimulated by a remote application. In the code illustrated in Appendix A, the variable "tcld_cmd" is utilized to receive commands by the daemon. This allows the command:

```
vwait tcld_cmd
```

to be utilized in a main loop to wait for incoming commands. Once the "tcld_cmd" variable is set by a remote application, a "switch" command:

```
switch $tcld_cmd
```

is utilized to select the proper routine.

Application Configuration

Now we will illustrate the application side of the Tcl-based interprocess configuration system. The application has three basic components: 1. a procedure for connecting to the Tcl

daemon, 2. calls to that procedure to read information from the daemon and 3. calls to that procedures to write commands to the daemon. Appendix B illustrates sample code for each of these components.

1. Daemon Connection Procedure

The daemon connection procedure ("send_tcld" in Appendix B) is used to send one string to the Tcl daemon. In this example a socket connection to the well known port of the Tcl daemon (2742 in this example) is made on the same machine:

```
catch {set tcld_channel [socket localhost 2742]}
```

Note that since the sockets utilize TCP/IP connections, remote connection on a different machine could be utilized by replacing "localhost" in the command above. The "fconfigure" command is utilized as in the daemon to configure the socket for line oriented communication:

```
fconfigure $tcld_channel -buffering line
```

The key components of the routine are sending the string to the remote host:

```
catch {puts $tcld_channel $tcld_string}
```

receiving the result string:

```
catch {gets $tcld_channel result}
```

then the socket is closed:

```
catch {close $tcld_channel}
```

and the result is returned to the calling procedure:

```
return $result
```

The sample code illustrated in Appendix B includes other error checking code. Also, it may be desirable to include diagnostic and logging code.

This implementation of the application code establishes and tears down a connection for each command / response sequence. It would alternatively be straightforward to have a persistent connection between the application and the Tcl daemo. Such a connection would stay open and each time a command string was passed to the daemon it would return a response string.

2. Daemon Read Procedure

The daemon read procedure ("getpos" in Appendix B) is used within the application code to read information from the Tcl daemon. The following command string is sent to the Tcl daemon:

```
set retval $pos_x $pos_y
```

This will return the two requested variables in a string or an empty list in the case of an error.

2. Daemon Write Procedure

The daemon write procedure ("sensors" in Appendix B) is utilized to send commands to the Tcl daemon. If the parameter "activate" is set, it will send the command string:

```
set tcld_cmd sensors_on
```

and otherwise will send the command string:

```
set tcld_cmd sensors_off
```

As explained above, setting this variable will trigger the execution of command code in the Tcl daemon. The result returned by this command will either be "OK" or "ERR".

Example Implementation

Figure 2 illustrates an example of how the inter-process communication mechanism described in this paper has been put into practice in an actual product. Three main Tcl daemons are utilized: a sensor daemon, a touch screen daemon and a plant network daemon. Additionally, a web server and a collection of CGI scripts are utilized. All of the daemons and the CGI scripts are written in Tcl.

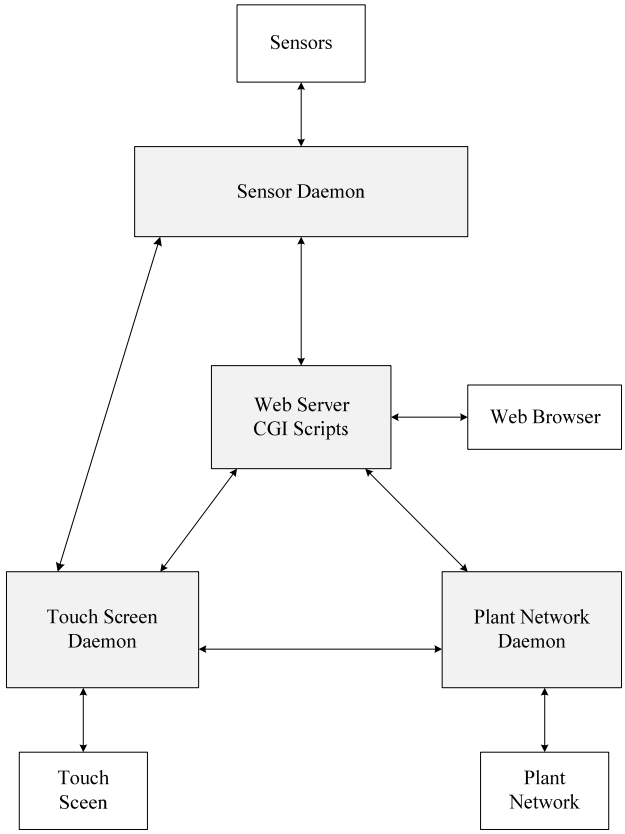

Figure 2

The sensor daemon is responsible for polling sensor units and maintaining information about their status and data collected. The sensor daemon is responsible for retrieving data from the sensors and to interpret it based on a selected configuration. Configurations are selected by the touch screen daemon or by the user through the web interface. The sensor daemon continually polls the specified sensors and computes device locations. Real-time device locations are communicated from the sensor daemon to the touch screen daemon. The socket between the sensor daemon and the touch screen daemon is a persistent connection while all other sockets are established and torn down on demand.

The touch screen daemon continually polls the sensor daemon for sensor information and provides it to the touch screen for display to the user. The touch screen daemon is responsible for interfacing with a touch screen to provide a user interface for monitoring and control. Through the touch screen the user can select a configuration and observe sensor information. The user can also cause a command to be sent to the plant network daemon for updating a manufacturing database coupled to the plant network.

The plant network daemon is used to handle communication with manufacturing databases. There is an information database that is queried to get information regarding the device currently being monitored and also a manufacturing database that is updated with sensed information.

The embedded system outlined in Figure 2 utilizes a single board computer running FreeBSD and running off of a CF card. Ethernet connections are utilized for communication with the touch screen, the plant network and the web browser. The sensors communicate via a proprietary CAN (Controller Area Network) based network.

Conclusions

In this white paper we have illustrated an application of inter-process communication mechanism based on Tcl daemons and application programs utilizing standard socket communication. These features utilize the core Tcl features and thus do not require any specialized packages. This implements an inter-process communication that is robust and is easy to test and debug.

The daemon and the application program are written in Tcl and communication takes place by sending Tcl command strings and returning Tcl lists. A predefined return value ("retval") and a command trigger variable (e.g. "tcld_cmd") are utilized to allow simple "set" commands to be used for all Tcl command sequences. The Tcl daemon is based on a polling loop to read device information and a main loop that responds to remote commands received. More complex daemon architectures can be implemented using this basic mechanism.

References

[1] Ousterhout, J. and Jones, K., Tcl and the Tk Toolkit, Addison-Wesley, Sept. 2009.

[2] "Inventory of IPC methods," http://wiki.tcl.tk/1228, updated Aug. 2011.

[3] Lidie,S. and Walsh, N., Mastering Perl/Tk, "Chapter 20. IPC with send," O'Reilly, Jan. 2002.

[4] "TCLLIB - Tcl Standard Library: comm," http://tcllib.sourceforge.net/doc/comm.html.

Appendix A – Sample Daemon Code

```
# Server invocations
#    Listen on 2742 for incoming connections
#
socket -server tcld_server 2742


# Incoming Connection Procedure
#   This procedure is called whenever a new connection is received on the incoming
#   socket used for internal connections.
#
proc tcld_server {channel clientaddr clientport} {

        # A new connection has been accepted
        # Configure channel for line-oriented buffering
        #
        fconfigure $channel -buffering line

        # Set up callback for when the client sends data
        #
        fileevent $channel readable [list tcld_read $channel]
}
#
# Internal Read Command Procedure
#   This procedure is called when data is received on the incoming socket
#   Received data consists of a single line Tcl command string.  The received string
#   is evaluated using the Tcl "eval" command.  If a return value is expected, the
#   command string should set a local variable named "retval".  If "retval" is not set
#   by the command "OK" is returned upon successful completion of the command and
#   "ERR" otherwise.
#
#   Every variable that a remote process needs access to must be declared global in
#   this procedure.
#
proc tcld_read {channel} {
        global tcld_cmd
        global pos_x
        global pos_y

        # Readable channel
        #   check end of file or abnormal connection drop
        #
        if {[eof $channel] || [catch {gets $channel line}]} {
                close $channel
        } else {
                if {[string length $line] > 0} {
                        #
                        # command string received, try to execute it
                        #
                        set retval "OK"
                        if {[catch {eval $line}] == 0} {
                                #
                                # command executed OK, return value and continue
                                #
                                puts $channel $retval
                        } else {
                                #
                                # error in execute of command string, return error
                                #
                                puts $channel "ERR"
                        }
                } else {
                        #
                        # empty line received, test EOF again
                        #   (observed empty strings just before socket close)
                        #
                        if {[eof $channel]} {
                                close $channel
                        }
                }
        }
}
```

```
# Polling Loop - Sample
#   This loop runs independent of incoming connections and inter process communication.
#   This sample code updates position variables once per second when active, in an
#   actual daemon the polling loop would be used for interaction with external devices.
#
proc poll {} {
        global pos_x
        global pos_y
        global active

        if {$active} {
                #
                # update pos_x and pos_y
                #
        }
        after 1000 "poll"
}


# Initialization
#   Initialize global variables and initiate first instance of polling loop
#
set pos_x 0
set pos_y 0
set tcld_cmd ""
set active 0
poll


# Main Loop - Command Processing
#   This loop is executed when an incoming command is received
#
while {1} {

        # Wait for a command to be sent by application program
        #
        vwait tcld_cmd

        # Get command
        #
        switch $tcld_cmd {
                "sensors_on" {
                        set active 0
                }
                "sensors_off" {
                        set active 1
                }
        }

        # Reset command and wait for next command
        #
        set tcld_cmd ""
}
```

Appendix B – Sample Application Code

```
# Send command to daemon procedure
#   This procedure is used to send strings to the daemon, it may be used for reading or setting
#   variables local to the daemon and to send commands to the daemon.
#
proc send_tcld {tcld_string} {
        global error_string

        set server localhost
        if {[catch {set tcld_channel [socket $server 2742]}]} {
                #
                # can't connect to server
                #
                set error_string "Can't connect to server"
                return ""
        } else {
                #
                # server connection OK, configure for line oriented buffering
                #
                fconfigure $tcld_channel -buffering line

                # send string to tcld daemon
                #
                if {[catch {puts $tcld_channel $tcld_string} error]} {
                        #
                        # error in sending to socket
                        #
                        set error_string "daemon socket error: $error"
                        catch {close $tcld_channel}
                        return ""
                }
                #
                # send OK, get response
                #
                if {[catch {gets $tcld_channel result} error]} {
                        set error_string "daemon socket error: $error"
                        catch {close $tcld_channel}
                        return ""
                }
                catch {close $tcld_channel}
                if {$result == "ERR"} {
                        return ""
                } else {
                        return $result
                }
        }
}


# Get position from daemon
#
proc getpos {
        global poslist;
        set poslist [send_tcld "set retval \"\$pos_x \$pos_y\""]
}


# Send commands to the daemon
#   Send “sensors_on” and “sensors_off” commands to the daemon
#
proc sensors {activate} {
        global result
        global error_string
        if {$activate} {
                set result [send_tcld "set tcld_cmd sensors_on"]
                if {($result != "OK")} {
                        set error_string "Unexpected result from daemon: $result"
                }
        } else {
                set result [send_tcld "set tcld_cmd sensors_off"]
                if {($result != "OK")} {
                        set error_string "Unexpected result from daemon: $result"
                }
        }
}
```