Advanced Programming in Expect: A Bulletproof Interface

by David L. Fisher

http://linuxgazette.net/issue48/fisher.html
http://dlf.cotse.net/papers/A_Bulletproof_Interface.pdf

Links to the scripts in this article are included in the Linux Gazette. Off-site links to Expect manual pages are indicated by a "(*)" after the link.

Introduction:

This article assumes the reader has a thorough understanding of the basics of the Expect scripting language and is looking for advanced solutions. For more on Expect, see:

> http://www.cotse.com/dlf/man/expect/index.html (*)

In the design of automated systems using the Expect programming language, one of the more difficult hurdles many programmers encounter is ensuring communication with ill-behaved connections and remote terminals. The **`send_expect`** procedure detailed in this article provides a means of ensuring communication with remote systems and handles editing and rebroadcast of the command line. Where a programmer would usually send a command line and then expect the echo from the remote system, this procedure replaces those lines of code and provides the most reliable interface I have come across. Features of this interface include:

- Guarantees transmission via remote system echo
- Tolerates remote terminal control codes and garbage characters in the echo
- Persistence of attempts and hierarchy of methods before declaring a failure
- Interactively edits and retransmits command lines that cannot be verified
- Maintains its own moving-window diagnostics files, so they are small and directly associated with the errors

Communication with local processes (i.e. those running on the same workstation as the Expect process) is typically not problematic and does not require the solutions detailed in this article. External processes, however, can create a number of problems that may or may not affect communication, but will affect an automated system's ability to determine the success of the communication. In cases where it is corrupted, it is not always immediately obvious: a corrupted command may trigger an error message, but data which has been corrupted may still be considered valid and the error would not show up immediately, and may cause a variety of problems. This is why it is necessary to ensure that the entire string that is transmitted is properly received echoed by the remote system.

The basic idea of this interface is to send the command string except for its terminating character (usually, a carriage return) and look at the echo from the remote system. If the two can be matched using the regular expressions in the **`expect`** clauses, then the terminating character is sent and transmission is considered successful. If success cannot be determined, the command line is cleared instead of being sent, and alternative transmission modes are used.

In many cases, nothing more than expecting the exact echo of the string is sufficient. If you're reading this article, though, I suspect that you've encountered some of the problems I have when programming in Expect, and you're looking for the solution here. If you're just reading out of interest, the problems arise when automating a session on a machine off in a lab, or on the other side of the world. Strange characters pop up over the connection, and the terminal you're connected to does weird things with its echo, but everything is working. It becomes very difficult to determine if what was sent was properly received when you have noise on the connection, terminal control codes inserted in the echo, and even server timeouts between the automation program and the remote session. This interface survives all of that, and if it can't successfully transmit the string, it means that the connection to the remote system has been lost.

The code provided in this article is executable, but needs to be incorporated into any system in which it is to be used. Ordinarily, system-dependent commands need to be added based on the needs of the target system. Also, this code uses simple calls to the **puts** (*) command to output status messages - these should be changed to use whatever logging mechanism is used by the rest of the system. A final caveat, and I can't emphasize this enough: always wear eye protection.

Setting Up The Interface:

The procedures provided in this article are:

- **send_expect_init**
- **send_only**
- **send_expect**
- **send_expect_report**

The interface is initialized with the **send_expect_init** procedure, which sets up all the **globals** required by the other procedures. See the section on controlling the behavior of the interface for an explanation of the parameters. The **send_expect_init** procedure is run once, at the beginning of execution (before the interface is to be used). It may be run a second time to restore settings, if necessary.

The **send_only** procedure is a wrapper for the **exp_send** (*) command, and is used by **send_expect** to transmit strings. The only time this procedure is called directly is for strings that are not echoed, such as passwords, and multibyte character constants, such as the **telnet** break character (**control-],** aka **^]**).

The **send_expect** procedure is the actual interface between the automated system and its remote processes, and is detailed in the next section.

Finally, the send_expect_report procedure is used at the end of execution to output the statistics of the interface for debugging. This procedure may also be run during execution, if incremental reports are needed.

Using The send_expect Procedure

Once the interface has been initialized using **send_expect_init**, and a process has been **spawned**, it is ready to be used with the syntax:

```
send_expect id command;
```

where

id = the spawn id of the session on which to send the command
command = the entire command string including the terminating carriage-return, if any.

This syntax, and the implementation of the expression-action lists, support multiple-session applications.

Many people who follow the documented examples tend to write the same kind of error-prone code, because they follow the example as if it's the best example, instead of just a simple example. Examples are kept uncluttered by the little details that make the difference between bulletproof code and code that will eventually fail. The examples provided in this article are simple examples but with more attention to detail, and where warranted a complete implementation is provided as an example. The **send_expect** procedure usually replaces only two lines of code in an existing system.

The full syntax for properly using the interface is actually:

```
if { [send_expect $id $command] != 0} {
    ## handle your error here
}
```

How It Works:

The interface uses four different transmission modes, in order:

- send the entire string and hope for the best (fastest, but least reliable)
- send the entire string using the **send_slow** (*) list
- send the string in blocks of eight characters
- send the string one character at a time (slowest, but most reliable)

If a mode fails, the command line is cleared by sending the standard control-U, the expect buffer is cleared, and the next mode is tried. Each mode except the last one can also have a failure tolerance set, using:

```
sendGlobals(ModeXFailMax), where X is either 1,2 or 3.
```

If this max value is set to a positive number, once the failures for that mode exceeds this value, it is no longer used. If it is set to 0, then each mode is tried for each transmission, regardless of the number of failures. Each of the modes uses the **send_only** procedure as a wrapper for **exp_send**. If this procedure returns an error, it most likely means that the connection was lost, and the **spawn_id** is checked to see if the session is still active. The error is returned to **send_expect**, which in turn returns an error to the calling procedure.

For local processes and robust remote connections, mode 1 is usually sufficient. If the remote system is a bit slow, mode 2 may be required. Mode 3 has proven invaluable when connected to routers and clusters which provide rudimentary terminal control. Mode 4 is rarely required, but acts as a backup to mode 3.

Moving Window Diagnostics

Expect provides a means of controlling the output of its internal diagnostics and expression-matching attempts using the command "**exp_internal**" (*). The **send_expect** interface makes use of this command to create a diagnostics output file for each transmission attempt - for each attempt, a new diagnostics log file is created using **exp_internal -f**. If transmission is successful, the file is deleted. If it fails, the file is renamed using the syntax

```
send.i.n.command.diags
```

where

i = the spawn id of the channel that had the failure
n = the number of the failure
command = the first word of the command string that failed to get sent properly.

If you've ever read a 30 Megabyte log file with all of the diagnostics from the beginning to the end of execution, you'll understand why this is necessary. The diagnostics files created using this method are usually less that 2 Kilobytes, and since they are directly associated with failures (because the window file is deleted for successful transmissions), debugging is far more efficient.

The moving window diagnostics file is the fastest and smallest way to implement full diagnostics output during the execution of a send command. If the transmission succeeds, this file is deleted. If there is a failure, this file is closed and renamed, and on the next invocation of the send command a new file is created. This results in very small files (comparatively) with all of the diagnostics from expect and the user-defined messages, from the very beginning of the attempt to send the command.

Ideally, if there are no failures during execution, there should be no more than one send diagnostics file in existence at any time, named **send.diagnostics**. If there are diagnostics files, each is associated with a particular failure and should be used in debugging that failure.

Controlling The Behavior Of The Interface:

The **sendGlobals** array contains all of the parameters used by the interface, and may be modified at runtime to control how the interface works. This section will cover the meanings of these parameters and how they may be modified. See **send_expect_init** for the initial values of these parameter.

The failure limit elements (**Mode1FailMax**, **Mode2FailMax**, and **Mode3FailMax**) determine how many failures are permitted for modes 1, 2 and 3 (respectively). A value of zero disables this limitation, and any positive integer sets the maximum number of failures for that mode before it is no longer used by the interface. There is no failure limit for the last mode.

The element **useMode** allows the system to determine which transmission mode should be used first, so that the less reliable modes (the first and second) can be bypassed. Allowable values for this parameter are 1, 2, 3, or 4. Invalid values will be replaced by the default mode (1).

If transmission errors are not considered fatal, the **sendErrorSeverity** element may be specified to a more tolerant value. Note that this parameter is not used internally, so if the automated system does not access this value, it won't affect the interface.

The **kill** element defines the command line kill character, which is defaulted to the Gnu-standard **control-U (^U).**

The **diagFile** parameter names the temporary internal diagnostics file (generated from **exp_internal**).

The **logDiags** allows disabling of all diagnostics output for faster execution, but be forewarned that disabling this feature well make debugging much more difficult.

The **sleepTime** parameter, when set to a positive integer, causes the interface to sleep for the designated amount of time before starting transmission. This is useful if the automated system appears to be going faster than the remote system can handle - the consistent loss of characters in the transmission phase usually indicates a speed and synchronization problem, and this parameter is provided as an allowance for such cases.

The **interval** and **delay** elements represent the two items in the **send_slow** list, which is used by the second and third modes.

For experimentation purposes, it is recommended that these parameters be modified by the automated system at runtime, rather than directly editing the defaults in the initialization procedure. Once valid settings are found the defaults may be changed to reflect them.

Failures:

When the **send_expect** procedure returns a failure, it indicates an unreliable connection to the remote system, and manual verification will confirm this. Such a failure is fatal to the reliability of any automated system, and must be corrected before the system can run properly.

If the procedure itself appears to be malfunctioning, the diagnostics files that were created during the failure should help in debugging. This interface has yet to fail with a reliable connection.