Kibitz – Connecting Multiple Interactive Programs Together

Don Libes

National Institute of Standards and Technology, Gaithersburg, MD 20899, USA

Abstract

Expect is a programming language for automating interactive programs. Recently, people have begun using Expect to connect *multiple* interactive programs together, allowing for new classes of applications. With some basic building blocks and a little scripting, it is possible to build such applications quickly.

This paper discusses the general technique, while focusing on a particular example: *Kibitz*. Kibitz connects multiple sessions and applications together, providing a means for consulting, group editing, or other cooperative tasks. Kibitz in turn, can be used as a module in building additional programs of this type. Using Kibitz, we demonstrate how to enable cron or background processes to call upon and interact with users, e.g., for guidance or debugging.

Due to program reuse, our approach avoids many portability issues already addressed and solved by existing programs. Kibitz has no special coding for byte swapping, structure encoding, or job control, and requires neither kernel modifications nor setuid or other permissions even though it runs over a network and deals with multiple users, job control, and sophisticated programs such as shells and full-screen editors.

Keywords: Automating interaction; Expect; Interprocess Communication; Software reuse; Tcl

Reprinted from *Software – Practice & Experience*, John Wiley & Sons, New York, NY, Vol. 23, No. 5, May 1993.

May 24, 1993

Introduction

Many programs were designed to be used only interactively. Yet people inevitably want non-interactive, automated use. Witness the numerous variants of ftp, each with capabilities (background ftp, recursive ftp, mirrored ftp, etc.) not performed by the ubiquitous interactive version.

The usefulness of connecting *multiple* interactive programs is becoming increasingly evident, although such applications are hard to recognize because of a traditional mind-set. For example, the *Frequently-Asked-Questions* list [1] from the Usenet newsgroup *comp.unix.questions* addresses the question "*Is there a program to monitor another user's terminal?*" with an answer that suggests either a hard-wired tap or programs that require kernel modifications to peek into non-portable data structures [2].

Kibitz is a script written in Expect [3][4] that connects a process to the keyboards and screens of two (or more) people. An obvious use is to allow one person to see what another is typing and receiving. More sophisticated uses are possible, such as allowing several people to edit the same document simultaneously.

Solving the Problem by Reuse of Existing Programs

To create Kibitz and similar programs, we wanted to write as little new code as possible – striking a balance between code reuse, portability, and efficiency. The UNIX philosophy encourages building larger programs out of smaller ones, but interactive programs are not traditionally considered as building blocks.[1] With Expect as an implementation base, we were able to incorporate interactive UNIX tools with non-interactive ones.

Expect is a tool for automating interactive programs. Like a movie script, Expect scripts also describe dialogues, but between a human and (possibly multiple) programs. Expect includes a programming language so that scripts can take different paths through dialogues. By programmatically describing these dialogues, it is possible to automate interactive programs.

With Expect, Kibitz uses a number of interactive programs – all without change. For example, Kibitz uses the interactive UNIX utility **`write`**[2] to send a message to a user. In contrast, **`talkd`**, the UNIX talk daemon, has its own dedicated **`write`**-like code wired-in.

Reusing interactive programs such as **`write`** allows significantly decreased development time as compared to programming from scratch. The time to develop Kibitz was one week. We estimate that without the code re-use techniques described here, it would have taken several months to produce a program of comparable quality.

Kibitz is based on a short Expect idiom for connecting multiple sources/sinks together. Figure 1 is an example that connects the output of one process to the input of another, and vice-versa. The scripting language is Tcl [5][6].

---

1. The experiences described in this paper come from a UNIX environment but apply to many other environments. Unless otherwise referenced, specific commands mentioned are found in any UNIX environment and described in the UNIX manuals.

2. **`write`** is not interactive in the sense that it prompts for information. Yet, **`write`** can only be started from an environment with an associated terminal. This behavior cannot be disabled and hence it is considered "interactive".

```
while {1} {
      expect {
            -i $process1 -re .+ {
                  send -i $process2 $expect_out(buffer)
            }
            -i $process2 -re .+ {
                  send -i $process1 $expect_out(buffer)
            }
      }
}
```

Figure 1. Expect fragment to connect one process to another

This fragment is a loop in which the **expect** command waits for input from either process. Processes are introduced with the "**-i**" flag followed by the descriptors, which in this script are **process1** and **process2**. (In the language, a "**$**" is used to distinguish variable references from literals.) If no "**-i**" flag appears, interaction occurs with the process identified by the variable **spawn_id**. This allows scripts dealing with only a single process to omit the constant references to the process descriptor. Processes are started by the **spawn** command, which also sets **spawn_id** as a side-effect.

The "**-re**" introduces a regular expression that must match the input in order to successfully complete the command. When the expression is matched, an associated action is executed. In this case, the **send** action sends the characters to the other process. The regular expression "**.+**" matches one or more characters, allowing the program to dispatch as many characters as it received in a single iteration.

Figure 2 is a complete script that starts two chess programs and then sends the output of each to the other. It uses the same idea as the earlier idiom except that this particular chess program[3] outputs some extra formatting that must be stripped off before it is acceptable as input. The script is further complicated because the formatting is different depending on whether the program moves first or second. These problems demonstrate additional reasons why interactive processes can be difficult to use as building blocks.

```
spawn chess;            set id1 $spawn_id
expect "Chess";         send "first\r"
expect -re "1. (.*)\n"; # read first move

spawn chess;            set id2 $spawn_id
expect "Chess";         send $expect_out(1,string)

while {1} {
        expect {
                -i $id2 -re "\.\. (.*)\n" {
                        send -i $id1 $expect_out(1,string)
                }
                -i $id1 -re "\.\. .*\. (.*)\n" {
                        send -i $id2 $expect_out(1,string)
                }
        }
```

3. Ken Thompson wrote the chess program which continues to be distributed with most versions of UNIX.

```
}
```

Figure 2. One chess process plays another

Kibitz uses a slightly more complicated version of the idiom. Figure 3 is a fragment that copies characters between two users and a shared process. This script interacts with three sources:

```
while {1} {
        expect {
                -i $user1 -re .+ {
                        send -i $process $expect_out(buffer)
                }
                -i $user2 -re .+ {
                        send -i $process $expect_out(buffer)
                }
                -i $process -re .+ {
                        send -i $user1 $expect_out(buffer)
                        send -i $user2 $expect_out(buffer)
                }
        }
}
```

Figure 3. Kibitz excerpt that connects two users to a common process

user1, **user2**, and **process**.

user1 refers to the first user's input. When user input arrives, it is sent to the process referred to by **process**. **user2** is the second user's input. It is similarly sent on to **process**. Thus, both users effectively control the same process.

The output of **process** is sent to both users. If they are interacting with a process that echoes keystrokes, both users see the results of any keystrokes typed by either user.

As before, all cases use the regular expression "**.+**" to wait for one or more characters of input.

Expect automatically provides a descriptor corresponding to the I/O of the user who invoked Kibitz. Here, that descriptor is called **user1**. By default, Kibitz sets **process** to be the I/O of the command "**spawn $env(SHELL)**" which starts a shell named by the user's environment variable **SHELL**. The only remaining problem is how the descriptor **user2** is created. This is described in the following two sections.

Intra-host Communication

Communication with a second user requires a two-way communications path (or two one-way paths). Writing C code to do this would have required a lot of work, much of it dealing with system-specific issues, such as port numbers, protocols, servers, etc. Of course, existing programs do create and use such communications paths, however they make it inaccessible to other applications because these communication capabilities are bundled with others.

For example, **talk** is a UNIX tool that performs user-to-user communications. **talk** also insists upon graphically formatting the conversation. This is inappropriate for processes (such as shells) that have no interest in receiving user commands interspersed with formatting characters.

telnet, on the other hand, does no formatting, but has a different problem. **telnet** requires one side of the connection to exist before entertaining requests for connections. While this bootstrap problem can be solved by the existence of a daemon, writing one would have led to extensive coding that seemed pointless when it was possible to reuse existing code.

Our solution uses fifos[4] to connect the original Kibitz process to a Kibitz process started by the second user. Amusingly, this represents the largest part of the script dedicated to portability. Because **mknod** (the UNIX program to create fifos) is not located in a standard place in the file system and is often not on users' paths, the script must explicitly search for it (Figure 4).

```
proc mknod {f} {
	if 0==[catch {exec /etc/mknod $f p}] return      ;# AIX, Cray, SG
	if 0==[catch {exec /bin/mknod $f p}] return      ;# Ultrix
	if 0==[catch {exec /usr/etc/mknod $f p}] return  ;# Sun
	send_user "Failed to make a fifo - where is mknod?\n"
	abort
}
```

Figure 4. Tcl procedure to create a fifo

The second user starts Kibitz when prompted (via **write** as mentioned above). The prompt includes a unique identifier to avoid conflicts with other Kibitz sessions. This unique identifier is the process id of the original Kibitz process. It is not necessary for the second user to appreciate this as the prompt specifies exactly what to type. The second user sees something like "**Message from susie@nephron on ttyp1: Let's talk. Please type: kibitz -709**".

A graphical view of the resulting process relationship is shown below.

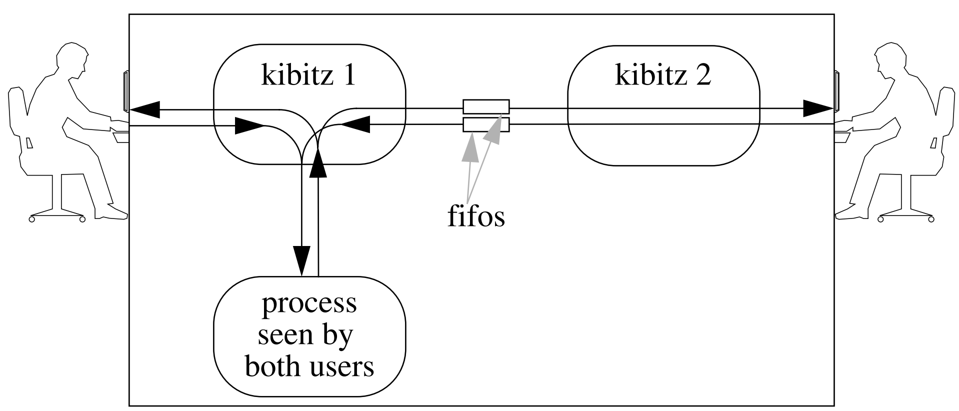

Figure 5. Kibitz session between two users on one host

Inter-host Communication

If the second user is not on the same system, it is necessary to cross machine boundaries. To avoid building in all the machinery for handling networking, Kibitz creates a communication path via **rlogin**. **rlogin** is used because it already understands hostnames, and it provides security (via **.rhosts**) and reliable transport. **telnet** could be used as well although **rlogin** normally

4. A *fifo* (for First-In-First-Out) is a UNIX abstraction of a one-way communications path.

works without passwords, and is thus preferred. If the remote system prompts for a password, Kibitz intercepts and rewords the prompt so that the user understands it is for the remote system. This is necessary because the rest of the interaction is not seen by the user, and a bare prompt for a password could be confusing to anyone unfamiliar with how Kibitz works internally.

While waiting for a password prompt, Kibitz also recognizes the command prompt from the user shell. Since different shells prompt differently, Kibitz uses a regular expression that matches a variety of common prompts. Kibitz will not send commands until matching the prompt.

Some users customize their prompt so that it is no longer matched by the default prompt pattern. In this case, users can supply a pattern to distinguish their prompt from other things printed at login such as a message-of-the-day. Recognizing prompts is a common obstacle when automating interactive applications. However, by storing the pattern in an environment variable, any program automater can use it. The prompt pattern definition is not specific to Kibitz.

Once the remote login is established, both users logically appear as if they are on the same machine. A Kibitz process is started on the remote machine to initiate the rendezvous, using the previously described method of communication via fifos. Except for the possible prompting for a password, starting and connecting all of these processes is invisible to the users. The result is illustrated in Figure 6.

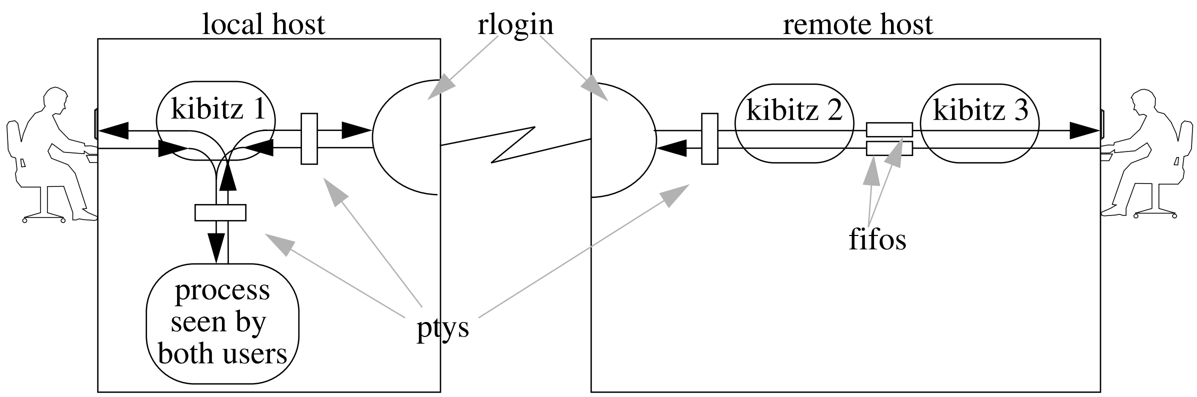

Figure 6. Kibitz session between two users on two hosts

Figure 6 also displays where pty (pseudo-terminal) interfaces are inserted. A pty is an interprocess interface that provides the semantics of physical terminals, enabling correct handling of screen size and other attributes. Expect, itself, inserts the pty interfaces automatically as Kibitz starts the shared process and the rlogin. For its own purposes, rlogin inserts the remaining pty.

If any errors occur before the remote Kibitz is started (e.g., “no such computer exists by that name”), the original Kibitz detects this and reports it back to the user. Normally, the original Kibitz only passes characters between the processes as indicated by Figure 6. However, it is possible for the remote Kibitz to need to report problems during initialization (e.g., “no such user exists by that name”). It would be inappropriate for the remote Kibitz to pass such error messages back to the original Kibitz only to have them sent to the shared process. Rather, the original Kibitz should send them only to the user who originally invoked Kibitz.

To avoid this problem, a tiny in-band protocol is used between the first two Kibitz processes. Once the remote Kibitz is running, it sends the string “**`KRUN`**”. This tells the original Kibitz that the remote Kibitz has begun running. Whatever the remote Kibitz sends now (presumably diagnostics) is sent directly back to the user by the original Kibitz. If the remote Kibitz fails to establish the connection, "**KABORT**" is sent to the original Kibitz that, having already passed on any diagnostics to the user, exits as does the remote Kibitz. If the remote Kibitz successfully establishes the connection, it sends back "**KDATA**" meaning that anything after this is user data and should be sent back to the shared process.

Beyond establishment of the process connections, the only thing remaining is to to define an escape mechanism (Figure 7). After entering the escape sequence, the user speaks directly to the Expect interpreter and can then perform job control or other out-of-band communications and operations. This interaction style exactly mimics the ubiquitous telnet user interface.

```
send_user "Escape sequence is $escape_printable\n"
expect_before -i $user_spawn_id $escape {
      send_user "to exit kibitz, enter: exit\n"
      send_user "to suspend kibitz, press the appropriate job control"
      send_user " sequence\n"
      send_user "to return to kibitzing, enter: return\n"
      interpreter
      send_user "returning to kibitz\n"
```

Figure 7. Define an escape mechanism for the user

After successful communications, either Kibitz process closes down the connection by local user request or upon reading an end-of-file from the other Kibitz.

Other Applications of this Approach

Kibitz provides a model for similar applications that connect multiple interactive programs. For example, we have experimented with a replacement for the UNIX **learn** program. The new version oversees users' interactions much more closely than the original. For instance, job control via ^Z, **fg**, **bg**, etc., can now be taught since the script can watch every keystroke without losing control, no matter what the user does. Similarly, **telnet**, shell-style history, and full-screen editors can be taught.

Another application is regression testing, a technique of comparing program behavior before and after a program has been modified. Regression testing is difficult to apply to interactive programs [7], especially with respect to user interfaces. It is our experience that user interfaces are rarely specified – **ftp** is a good example of this. The **ftp** protocol is well-defined [8]; the user interface is not. Using the techniques described here, it is possible to write a script that concurrently tests two (or more) versions of the same program while a user puts them through their paces in tandem. The script can record the entire interaction with notations about where they differ. At any time, the user could change to viewing output from a different version.

Yet another application is to "borrow" the front-end of one program by another. For example, **ksh** [9], a UNIX shell, has a sophisticated user-interface that includes the ability to automatically complete filenames and edit the command line using either **emacs**- or **vi**-like commands. It is possible to have **ksh** syntactically and semantically analyze commands. When the user has completed composition of the command, it is passed to another program for evaluation. Meanwhile the **ksh** input buffer is cleared for the next command. This technique enables a simple program to behave as if it had a sophisticated user-interface. This can be helpful for deciding which of several different interface styles to adopt before actually selecting one to implement at great expense.

Using Kibitz as a Building Block

Kibitz is an interactive program itself. It may thus be re-used as a building block within the very technique described to build it. For example, the following Expect script (Figure 8) allows a **cron**[5] or background process to contact the user and request help. The "**-noproc**" option to Kibitz skips the creation of a new process with which to interact, providing instead a direct connection between two users, as suggested by Figure 1.

```
spawn some-process; set proc $spawn_id
. . .
. . .
# assume script now has a question or problem
spawn kibitz -noproc some-user
send "I've never seen this situation before, help!\n"
interact -u $proc
send "Thanks for your help\n"
```

Figure 8. How a background process can contact a user for assistance

When run from **cron**, the script requests help from a user. If the user acknowledges the request, the script can present a question or dialogue, or can hand complete control over to the user. One can imagine questions such as "*I need a password to continue*" or "*The 3rd backup tape is bad, replace it and tell me when I can go on*". This technique also provides a way to experiment within the **cron** environment and debug its processes *in situ* – something not otherwise possible.

The script works as follows. The first line runs an interactive program. Expect interacts with it for a while (indicated by ellipses) and eventually finds that it needs assistance. It attempts to contact a person by running Kibitz. If a person responds, the script explains the situation to the person. Although it is possible for a script to mediate access between the person and the process, this example script effectively joins them together with the **interact** command (Figure 9).

interact is an Expect command that connects the output from one program to the input of another and vice-versa. During the **interact**, the user is in essence talking directly to the troublesome process that was originally being controlled by the Expect under **cron**. When the user relinquishes control, both Kibitz processes go away and Expect continues executing its script.

How Well Does it Work?

This technique of connecting interactive programs together is very general, allowing arbitrary data transformation and midstream rearrangement of data flow.

---

5. A ***cron*** process runs with a user's privileges but is started by the system, usually at a pre-designated time. Unlike a more conventional background process, a **cron** process never has a controlling terminal and lacks certain other attributes found in a real user's interactive environment.

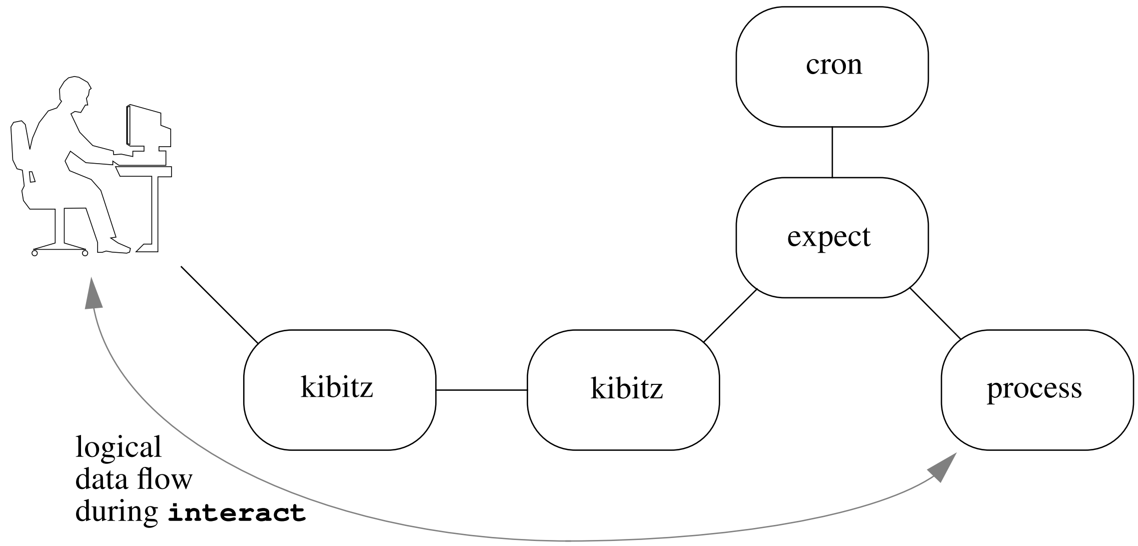

Figure 9. User providing assistance to a cron process

The specific case of Kibitz demonstrates a clean user-interface. The underlying interactive programs are entirely out of the user's view. Yet Kibitz follows traditional user expectations such as abiding by the **.rhosts** authorization model and providing shell-style job control. Because of the seamless transparency, users have been quite surprised to find out Kibitz is "just a script" joining old programs together rather than a brand new C program.

Kibitz aids communication from users who have a problem (or developers who have their solution) but cannot describe it correctly. Now, a developer can often assist a user immediately, instead of the two of them mailing transcripts back and forth. (Developer-to-user: "*Well it works on my account. What is in your environment, what directory were you in, what ...*")

Ironically, Kibitz is more flexible than **talk** for the function for which **talk** itself was designed – again, because Kibitz supports interaction with any process. **talk** does not. Connecting Kibitz to a full-screen editor allows a conversation to be formatted automatically – like **talk**. Unlike **talk**, users can also edit the conversation, save it, read in other files, etc., while carrying on a discussion. All of this functionality is provided by whatever interactive process users choose themselves.

When it comes to networked connections, the approach described here has a drawback that the initiating user must have an account on both machines. It is possible to remedy this by using a server. On the other hand, the likelihood of users Kibitzing between unrelated hosts is extremely low. After all, Kibitz effectively allows the second user to masquerade as the first user (in the context of the particular application). Thus, a significant amount of trust must exist before one user attempts to Kibitz with another, even on the same host, no less a different one.

Another limitation of Kibitz is that it cannot be applied to an existing data stream. Kibitz must create new streams. This can be paramount in debugging certain kinds of problems. In contrast, **advise** [2] can be applied to existing data streams, although it is highly non-portable requiring kernel modifications and does not work across a network.

Programming Effort Required

We have written a number of scripts using the techniques described here. Based upon this experience, we took one man-week to create Kibitz including debugging and documentation. The production script for all of Kibitz is 385 lines. About half of the code is dedicated to handling error conditions and a third to comments.

# of lines	function
105	comment or blank lines
70	copy loops
35	process creation, control, and cleanup
25	fifo and pipe creation and cleanup
25	domain name
25	inter-process handshaking
30	argument processing
30	informational user messages

The remaining lines handle the usual overhead such as appropriately setting signals, terminal modes, etc. The twenty-five used by "domain name" reflects the lack of high-level commands to decide whether (possibly partially qualified) hostnames are equivalent to given local host and domain names, or for that matter, what the local names are. For example, the UNIX command **`domainname`** returns the NIS name on many hosts, so the Kibitz code must do some digging in order to obtain the Internet domain name. **`hostname`** is similarly nonstandard in what it returns.

Using Kibitz as a building block allows scripts requiring very few lines of code. Except for the missing application-dependent code, the earlier six-line script (Figure 8) is complete.

While Expect is an interpreter, significant effort has been invested in efficiency at each level in the system. For example, commands and variables are hashed in the underlying Tcl interpreter. Expect itself compiles commonly-used regular expressions. And the actual Kibitz code has minor refinements that increase performance by further reducing interpretation (e.g., the **`while`** and **`expect`** statements in the loop shown above are compiled). Size and performance statistics on Expect itself were reported previously [3].

Conclusion

This paper has provided motivation and described techniques for connecting multiple interactive programs. The techniques themselves are straightforward, portable, and substantially reduce programming time.

Kibitz and the other examples mentioned solve long-standing problems in the UNIX environment. At the same time, the effort invested in writing any of them is quite small, due to the extensive reuse of existing interactive programs.

Acknowledgments

This work was supported by the National Institute of Standards and Technology (NIST) Automated Manufacturing Research Facility (AMRF). The AMRF is funded by both NIST and the Navy Manufacturing Technology Program.

Joe Petolino of Sun Microsystems demonstrated to me the technique of front-end borrowing described here. Corey Satten of the University of Washington wrote a Kibitz server to demonstrate that it is possible to avoid the restriction forcing the initiating user to have an account on both machines.

Availability

Since the design and implementation of Expect and Kibitz were paid for by the U.S. government, they are in the public domain. However, the author and NIST would appreciate credit if these programs, documentation, ideas, or portions of them are used. Expect and Kibitz may be **`ftp`**'d as **`pub/expect/expect.shar.Z`** from **`ftp.cme.nist.gov`**. Expect and Kibitz will be mailed to you if you send the mail message "**`send pub/expect/expect.shar.Z`**" (without quotes) to **`library@cme.nist.gov`**.

References

[1] Ted Timar, "Frequently Asked Questions about UNIX – with Answers [Monthly posting]", *comp.unix.questions*, <questions_694705327@nff.ncl.omron.co.jp>, Omron Corporation, Kyoto, Japan, posted monthly.

[2] Keith Gabryelski, "advise – Attach to another user", unpublished manual page, October 16, 1990, Commodore, Inc., West Chester, Pennsylvania.

[3] Don Libes, "Expect: Curing Those Uncontrollable Fits of Interaction", *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, California, June 11-15, 1990.

[4] Don Libes, "Expect: Scripts for Controlling Interactive Programs", *Computing Systems*, Vol. 4, No. 2, University of California Press Journals, November 1991.

[5] John Ousterhout, "Tcl: An Embeddable Command Language", *Proceedings of the Winter 1990 USENIX Conference*, Washington, DC, January 22-26, 1990.

[6] John Ousterhout, "Tcl(3) – overview of tool command language facilities", unpublished manual page, University of California at Berkeley, Berkeley, California, January 1990.

[7] Don Libes, "Regression Testing and Conformance Testing Interactive Programs", *Proceedings of the Summer 1992 USENIX Conference*, San Antonio, Texas, June 12-15, 1992.

[8] Jon Postel and Joyce Reynolds, File Transfer Protocol (FTP), RFC 959, Network Information Center, SRI International, Menlo Park, California, October 1985.

[9] Bolsky, Morris I., and Korn, David G., *The Korn Shell: Command and Programming Language*, Englewood Cliffs, NJ, Prentice Hall, 1991.