EMail With A Mind of Its Own:
The Safe-Tcl Language for Enabled Mail

Nathaniel S. Borenstein

First Virtual Holdings, Inc., 25 Washington Avenue, Morristown, NJ 07960, USA
Email: nsb@nsb.fv.com

Abstract

A uniform extension language for email systems can radically extend the utility of electronic mail, simplifying the construction of mail-based services and permitting the delivery of active messages that interact with their recipients and take differential actions based on the recipients' responses. This paper describes such a language, Safe-Tcl, including the strong security and portability constraints it has to satisfy, and outlines its fundamental design.

Keyword Codes: C.2.4, D.3.2, H.4.3
Keywords: Distributed Systems, Programming Language Classifications, Communications Applications

I. The Dream of Enabled Mail

The phrase "enabled mail" encompasses several technologies that share the common goal of significantly increasing the power and utility of electronic mail systems. Enabled mail, in general, is the augmenting of electronic mail systems by the introduction of computational power at several key points in the electronic mail process. Many existing systems offer such power in some places, but the power is incomplete, non-uniform, and suffers from a lack of interoperability across platforms and tools.

A conceptual model for the introduction of computational power in email systems is given in [13]. Specifically, the general email model is enhanced by viewing the delivery process as consisting of three distinct phases: First, "delivery-time", which occurs immediately before the message crosses the delivery slot, and in which the message is still conceptually under the control of the sender; Second, "receipt-time", which occurs immediately after the message crosses the delivery slot; and, finally, "activation-time", which occurs whenever the recipient processes the message. This model provides an abstract framework for considering various uses of computational technology to provide enabled mail.

The types of enabled mail most commonly found in existing system allow users to specify customized computational processes to be executed upon the receipt of a mail message, or upon the incorporation of a message into a user's message store. Various

mail systems offer extremely powerful languages for this type of functionality. Because this functionality is largely incorporated within a single software suite, the fact that each of these systems uses a different language and model is only a minor nuisance, particularly for those attempting to switch between different mail systems.

Less common, and more of a radical break for most users and implementors, is the notion of "active" mail. Traditionally, electronic mail has been a passive, unidirectional medium. Even multimedia mail [1, 2, 3] has operated this way -- mail messages contain text that is displayed to the user, images that are shown to the user, audio that is played for the user, and so on, but the process is traditionally one-way and non-interactive. In active mail, a message contains a program to be executed when the recipient reads the message. Such a message can potentially perform arbitrarily complex interactions, vastly increasing the scope and utility of electronic mail. Active mail has hitherto been largely confined to experimental research software, due to problems of security, portability and standardization. [1,7,8,9] Recent research [10] has demonstrated workable solutions to the security and portability problems, making the time ripe to consider the wider deployment and eventual standardization of a language for active mail. In the area of active mail, convergence on a single language is critical, because email that does not interoperate in a cross-platform manner is of extremely limited utility.

Recently, the author and a colleague, Marshall Rose, have published a draft specification of a model for enabled mail, clarifying the conceptual roles of the various places in the email process where computation can augment the email process. [13] Additionally, we have published a specification and public-domain implementation for a language, Safe-Tcl, that we believe can serve as a uniform, cross-platform engine for all of these kinds of mail-enabling computation [14]. While standardization is more important for some aspects of enabled mail (notably active mail) than for others, there is much to be gained from using a uniform language at all phases of the enabled-mail process. We have based our language on Tcl [4, 5, 6], a language that was explicitly designed to be embedded as a computational extension to a larger application. Aside from our highly subjective judgment that Tcl is a very well-designed language for such purposes, Tcl has the virtue of being simple, well-defined, and available in a high-quality, extremely portable public domain implementation.

The choice of Tcl as a base language provides the basic syntax and many of the primitives for an enabled mail language, but vanilla Tcl is not itself suitable for enabled mail. The design of a language for enabled mail involves a complex combination of constraints. For some parts of the computation, where the program comes entirely from a trusted source (such as the user on behalf of whom it is executing), any sufficiently powerful and well-defined language would probably suffice, especially if augmented with messaging-specific features. The most severe constraints come from the active mail case, where security, interface portability, and cross-platform availability are absolutely critical.

The remainder of this paper is structured in two parts. First, we outline the problems inherent in the design of a language for enabled mail. Then, we describe the Safe-Tcl language and how it solves these problems. Concluding sections discuss the current

status, availability and future prospects for the Safe-Tcl language in particular and enabled mail in general.

II. The Key Problems: Designing a Language for Enabled Mail

As stated previously, some types of enabled mail, such as programs to filter incoming mail, can be provided in almost any general-purpose programming language. To provide a uniform language for all phases of the enabled mail process, however, special attention must be paid to the "worst case" -- active mail. In active mail, any electronic mail user can send a program to any other electronic mail user, to be executed in the recipient's environment. This raises several severe constraints which must be addressed in the design of an enabled mail language.

Security

The most critical constraint for active mail is security. Simply put, it must be possible to read a message from your mortal enemy without that message doing you any harm. The use of an arbitrarily powerful programming language in this role would be completely unacceptable, as it would allow malicious users to send email messages that deleted the recipient's files, stole confidential information and mailed it elsewhere, forged email impersonating the email recipient, or caused any number of other kinds of mischief.

Fortunately, recent research [10] has demonstrated that a language for active mail can be suitably restricted and constrained to do no such harm. The details are beyond the scope of this paper, but the basic idea is quite simple: you simple remove from the language any features that can be used to do harm, and then augment the resulting, impoverished language with less general primitives that provide a safe subset of the removed functionality. For example, you might remove the general-purpose mechanisms for reading and writing files, but replace it with new primitives that only allow limited operations on a specific set of "public" files.

When the solution is framed in these terms, it is clear that, with sufficient attention to some subtle details, a language can be made safe for widespread active mail use. What is probably less clear is that such a language will retain sufficient power to be broadly useful.

Power

After an active mail language is made safe, the question of the language's power becomes important. Previous active messaging systems either left the burden of security entirely to the user [1], sidestepped it by restricting all computation to a trusted environment, which doesn't scale to the most general email networks [7, 9], or restricted the language so severely as to fundamentally prohibit certain classes of potential applications [10].

The fundamental problem is that the most dangerous primitives, where active messaging is concerned, tend to be the most general primitives. The process of making a language safe for active messaging consists largely of replacing general primitives with more specialized safe primitives. Inevitably, the resulting loss of expressive power handicaps the creation of certain applications, particularly those applications not anticipated by the language designers.

Extensibility

The obvious solution to the problem of insufficient power in an active messaging language is to simply add more specialized safe primitives as needed. This, however, is a challenging idea when applied to the whole universe of email systems. Upgrading all systems to support the new primitives, however safe they might be, will generally be so difficult that it rarely happens. The biggest innovation that Safe-Tcl has over previous active messaging languages (such as ATOMICMAIL) is its extension model, which makes it plausible to grow the system's functionality in a safe evolutionary manner.

Missing from all previous active messaging systems is any systemic support for the distributed maintenance and evolution of the set of known safe primitives. In all such systems of which the author is aware, the only way to extend the language in a manner that augments its fundamental power (as opposed to simply defining new procedures using the existing primitives, which provides conceptual and computational convenience but no real increase in power) is to recompile the language interpreter itself, typically a major enterprise. Ideally, an active messaging language should provide some simple mechanism by which users can define specialized operators in an unrestricted environment, and make those specializations alone available to active mail programs.

Authentication

Once the ability for users to provide power-augmenting extensions is provided, the issue of authentication inevitably arises. For many applications, users will want to define actions that are available only for active messages from certain trusted senders. Trusting the mail headers is dangerously naive in this regard, as mail is trivial to forge in most environments. This implies that the ideal active messaging language would have the ability to understand and validate authenticated mail (using such technology as Privacy Enhanced Mail (PEM) [11] or Pretty Good Privacy (PGP) [12]). Such capabilities have not been available in any previous systems for active messaging.

Interface Portability

The notion of portability is of obvious importance to a language for enabled mail, as it will have to run on the widest imaginable variety of computing platforms. When most people think of portability, however, they tend to think primarily about the issues involved in writing portable software. While these issues are as important to enabled mail as they are to any other multiplatform application, there is an even harder issue for active mail, that of user interface portability.

The problem is that the sender of an active mail message will not, in the general case, have any knowledge about the type of computing platform or platforms on which his message will be read. This means that he cannot write programs for a particular user interface model such as that provide by Windows, the Macintosh, X11, or a teletype. The program must be written and delivered without any knowledge about the environment in which it will interact with the user.

Previous research active mail systems typically ignored this issue entirely, generally focusing on providing a single-platform active mail, which was rightly perceived as "hard enough". The ATOMICMAIL system [10] was the first to provide any kind of solution to this problem. In ATOMICMAIL, all user interaction took place via abstract primitives such as "get a string from the user" or "ask the user a multiple choice question". The implementation of these primitives varied significantly on the supported platforms (Macintosh, X11, smart terminals, and teletypes), but this variation was invisible to ATOMICMAIL programs, which simply knew, for example, that they had asked the user a multiple choice question and somehow gotten an answer. ATOMICMAIL thus demonstrated that it was conceptually possible to write interactive programs that were entirely ignorant and independent of the user interface platform on which they would be executed.

Interface Quality

Unfortunately, ATOMICMAIL's solution to the problem of user interface portability led directly to another problem, that of interface quality. ATOMICMAIL was generally perceived as solving the portability program by making it possible to produce programs that would have a user interface that was uniformly terrible on a wide variety of platforms.

The problem was that ATOMICMAIL's abstractions constituted a Procrustean bed into which all interactive applications had to be made to fit. An ATOMICMAIL program might be running in the world's most sophisticated user interface environment, but it couldn't exploit that power in any way other than to use the world's most sophisticated implementation of "ask the user a multiple choice question". In particular, it was impossible to develop a modern event-driven program with this model.

To succeed with a broad base of casual users, an active messaging language must provide some way to write programs with better user interfaces, without sacrificing the portability demonstrated by ATOMICMAIL.

III. The Safe-Tcl Language

With a solid history of experimental and proprietary technologies demonstrated for various aspects of Enabled Mail, the time seemed right to begin designing a language that would be suitable for all the various aspects of enabled mail. Unlike previous research projects, the goal of this effort was not pure research -- although some interesting problems remained, as described in the previous section -- but rather the development of a language that was good enough to be a candidate for standardization.

That the project was begun with this in mind does not imply that the authors think the language should necessarily be standardized as it now stands, nor that it is impossible that a better candidate will appear. Rather, the goal was to produce an enabled mail language that was good enough to initiate a standards process in this area.

In this section we will describe the basic technology of the Safe-Tcl language. A detailed description of the language is beyond the scope of this paper, but may be found in [14].

Tcl and Tk: A "Good Enough" Language Model

The history of standardization efforts shows that there is no such thing as an uncontroversial or simple standard. Indeed, most people don't realize that even such a basic and near-universal standard as ASCII remains the subject of ambiguity and debate. (There are several slightly different things that are considered "ASCII" in different communities; sticklers for detail will always reference a specific standard such as [15].) However, of all the things one might wish to standardize, it is hard to imagine anything more prone to controversy than programming languages. Programming languages come in a very rich diversity and differ widely in their basic syntax, features, and philosophies.

Fortunately, the world does not, in general, need a single standard programming language; applications written in multiple languages can coexist happily on most systems. Unfortunately, in the area of enabled mail, and particularly for active mail, a standard language is essential. It would be unrealistic to expect that there could ever be a single language that satisfied all interested parties. Instead, the goal of the effort that produced Safe-Tcl was to produce a language that was "good enough" for essentially all uses of enabled mail. We knew that the choice of the base language would inevitably be controversial and that some people would hate any language we might choose. It is our hope that those who dislike the language model we settled on will recognize the importance of converging on a single such language, and will try to shape their criticisms constructively, helping to evolve the language by correcting whatever deficiencies they might perceive in it.

The basic language model we chose was Tcl, the Tool Command Language developed by John Ousterhout at the University of California at Berkeley. The great strengths of Tcl, for this application, are:

-- It has an extremely simple, easily learned syntax.
-- It is interpreted rather than compiled.
-- It was explicitly designed to be embedded in larger applications.
-- There is a high quality multi-platform public domain implementation available.
-- There is a high-level graphical toolkit (Tk) available for X11 programming.

Major drawbacks we perceived in Tcl were:

-- There is no high-level graphical toolkit for non-UNIX platforms. However, the elegant structure of Tk is convincing proof that such toolkits are eminently plausible.

-- The variable scoping mechanism offers no intermediate scope between local and global variables. (However, access to other evaluation contexts is available using the Tcl "uplevel" and "upvar" facilities, which makes most things possible, if not graceful.)

In the end, there was no candidate language that seemed a closer fit to our needs than Tcl, but we recognized from the start that the language would have to evolve before it could be standardized. This is precisely the process we sought to initiate.

MIME Types for Safe-Tcl

With the emergence of MIME [3] as the widely-implemented standard representation for multimedia mail, it was clear from the start that we wanted Safe-Tcl to be MIME-smart. We defined two new MIME content-types for Safe-Tcl.

The first, "application/safe-tcl", is the MIME content-type for an actual Safe-Tcl program. That is, a Safe-Tcl program may be included as data anywhere in a MIME message by labeling it with this content-type value. An additional parameter that may be provided on the content-type line is the "evaluation-time" parameter, which may have one of two values, "activation" or "delivery". If the evaluation-time is "activation", this means that theprogram is to be evaluated when the user reads the message, i.e. the message is an active mail message, as described above. If the evaluation-time is "delivery", then the program is intended to be evaluated in a non-interactive process at the time of final message delivery, i.e. when the message is delivered to the user's mailbox. In neither case are any unsafe primitives to be made available to the program. (It is, however, desirable to have a mode in which the Safe-Tcl interpreter makes the unsafe primitives available to a program that is not received in the mail, but is supplied by the user as part of his customization environment.) This reflects the basic enabled mail model set forth in [13].

The second content-type we defined is "multipart/enabled-mail". This is a new MIME compound type, which (as per the MIME standard) shares a common syntax with all other MIME compound types, but which has different semantics. A MIME object of type multipart/enabled-mail has exactly two sub-parts. The first of these is an arbitrary MIME object (of any content-type, including possibly another compound type). The second is of type application/safe-tcl. The intended semantics are that the first MIME entity is not displayed to the user, but rather is made available for the manipulation of the Safe-Tcl program.

One advantage of this approach is that if a message of type multipart/enabled-mail is read using a mail tool that understands MIME but does not understand the Safe-Tcl extensions, the user will be able to view all the sub-parts (pictures, sounds, etc.) even though the interactive structure is missing. This means that the sub-parts can be ordered and structured with such serial viewing in mind, possibly even including explanatory textual parts that are never actually used by the Safe-Tcl program when it runs, but are there only for serial-viewing by MIME readers that do not support Safe-Tcl. (Currently, such careful structuring must be done by hand, but tools to automatically generate this kind of message are perfectly plausible.)

Safe-Tcl Restrictions

Safe-Tcl may be described as an extended subset of basic Tcl. Certain Tcl commands were considered unsafe for use in active mail, and are not made available to untrusted programs. These include all commands that access files or execute system commands. Once all of these are removed from Tcl, however, the language has no remaining capability to interact with the user or the environment, so the addition of more limited commands for this purpose was essential.

Safe-Tcl Language Enhancements

The Safe-Tcl language supplements the restricted Tcl subset with a number of commands that give untrusted programs a limited ability to interact with the user and the environment.

Some Safe-Tcl commands are used to store and retrieve persistent data in a safe way. (The storage mechanism used is implementation-dependent, but will typically involve a single file or directory.) The information that one Safe-Tcl program stores in this manner is available to any other Safe-Tcl program; there is no mechanism for protecting one Safe-Tcl application from another, because there is no way of telling that messages A and B are not generated by the same application as message C. This situation could be remedied with authentication-smart extensions, using the mechanisms to be described below.

A great many Safe-Tcl commands deal with messaging-specific details such as address and date parsing, MIME object composition and decomposition, and so on. This gives language-level support for some of the most commonly-desired actions that users will want to take with Safe-Tcl.

Additional Safe-Tcl commands allow untrusted programs to send mail and print data, but only after the data in question is presented to the user and the user’s consent is obtained.

Another Safe-Tcl command is used to access a distributed library of Safe-Tcl extension programs. Such libraries may be obtained locally or from a user-authorized repository elsewhere on the network, but should only refer to sources trusted by the user.

Safe-Tcl Interface Styles

Most of the remaining Safe-Tcl commands deal with user interface issues. Safe-Tcl seeks to have the universality of interface of the ATOMICMAIL program, as described above, but without precluding the possibility of having Safe-Tcl programs actually be pleasant to use. This goal is achieved by the introduction of a notion of user interface styles.

Each Safe-Tcl user interface style is, in essence, a package of Safe-Tcl extensions that permit user interface programming on a certain platform or platforms. In particular, Safe-Tcl defines a user interface style of "Tk3.6" for Tk-based graphical interaction

under the X11 window system. Safe-Tcl also defines a user interface style of "generic" for user-interface-independent interaction, in the manner of ATOMICMAIL. The Safe-Tcl interpreter always provides a global variable, SafeTcl_InterfaceStyle, which contains a list of interface styles supported by the currently running Safe-Tcl interpreter. This list always includes "generic".

This means that, with the current system, one can write Safe-Tcl code that checks the locally available user interface styles, providing a high-quality graphical user interface if X11 is running, and providing a less pleasant but still usable interface if X11 is not running. Eventually, it is anticipated, there may be Safe-Tcl interface styles for Windows, Macintosh, or any other important platforms. (Eventually, it might also be desirable to define an abstract user interface style that is both system-independent and graphically-oriented, the higher-level analog of the "generic" style.) But any program that has a fallback to the "generic" case will be able to run with any Safe-Tcl interpreter. In this manner, Safe-Tcl provides a user interface that is no better than ATOMICMAIL's in the worst case, but considerably better in many other cases.

Safe-Tcl Multimedia Capability

Designing Safe-Tcl with MIME in mind made it almost trivial to give the language rich multimedia capability, a marked contrast with previous systems for enabled mail. In particular, the language includes primitives for getting a list of parts inside a multipart MIME object, and for displaying any particular subpart. The latter functionality is implemented by calling a general-purpose MIME-displaying tool. Our implementation works with either of two such interpreters, mhn or metamail.

The simple Safe-Tcl primitives for MIME access open up a rich range of multimedia behaviors, such as offering the user a button to click on to see a video clip, and makes Safe-Tcl suitable as the "control structure" for almost any interactive multimedia application. In fact, when the interpreter is run without the "safe" mode, restoring all the power of full Tcl while retaining the Safe-Tcl extensions, the result is an extremely powerful scripting language for arbitrary (non-mail-based) multimedia applications.

Safe-Tcl Extensions

Because Safe-Tcl replaces general Tcl primitives with safer but much less powerful primitives, it is inevitable that application writers will think of things that could be done safely, but only with the introduction of a new Safe-Tcl command. In previous systems, this would have been intractable, especially in a distributed environment, due to the sophistication needed to modify the language interpreter and even more importantly to the extreme difficulty of getting such extensions installed everywhere on the network.

Two aspects of Safe-Tcl's design make the problem much more tractable: twin interpreters and distributed libraries.

Because the basic Tcl language was designed to be embedded in larger applications, the implementation is extremely sensitive to the possibility that there might be multiple Tcl-based extension languages in a single process and address space. For this reason, Tcl's key data structures are not global, but are tied to an object known as a Tcl interpreter. Multiple Tcl interpreters in a given process can implement different Tcl-based extension languages without interfering with each other.

The implementation of Safe-Tcl relies heavily on this feature, and is based on a "twin interpreter" model. The Safe-Tcl process contains two interpreters, a trusted interpreter and an untrusted interpreter. Untrusted code (such as an active mail message) is evaluated in the untrusted interpreter, which is modified in all the ways described above. The trusted interpreter never directly evaluates untrusted code, but may be used to extend the untrusted interpreter. This is done by defining a procedure in the trusted interpreter, and then exporting it to the untrusted interpreter. This effectively makes a new bit of functionality available to untrusted programs without actually modifying the underlying C program that interprets the Safe-Tcl code. As long as the extensions are carefully designed and implemented, this kind of extensibility does not compromise security, although it does give users the ability to "shoot themselves in the foot" by inadvertently creating unsafe extensions. (An important thing to avoid, for example, is having the extension code evaluate its arguments as Tcl subprograms, which would effectively import untrusted code into the trusted interpreter.)

The relationship between the two interpreters is one of the most subtle and important aspects of the safe-tcl system. In order to use the system wisely and safely, it must be understood well by extension programmers. A good analogy from the domain of timesharing operating systems is the common distinction between "user space" and "kernel space". The trusted interpreter operates in an unrestricted environment, similar to the kernel in a UNIX operating system. The restricted environment of the untrusted interpreter is analogous to user space, where a more limited set of capabilities are available. In this analogy, each time the unrestricted interpreter adds a new bit of functionality to the restricted interpreter, it is performing the analogue of defining a new system call to the kernel.

The twin interpreter model makes it much easier to safely extend the functionality of the restricted Safe-Tcl language, but does not address the problem of distributing extensions on the network. For this, Safe-Tcl provides a primitive called "SafeTcl_loadlibrary". This primitive will attempt to find and load a specified Safe-Tcl extension package. By default, the interpreter will only look in the local Safe-Tcl libraries for such extensions, but the system can be configured to download extensions from a trusted FTP site as well. Thus a community of users who share trust in such a repository can all have their Safe-Tcl interpreters simultaneously extended by the addition of an extension file to an FTP repository.

Authentication and Safe-Tcl

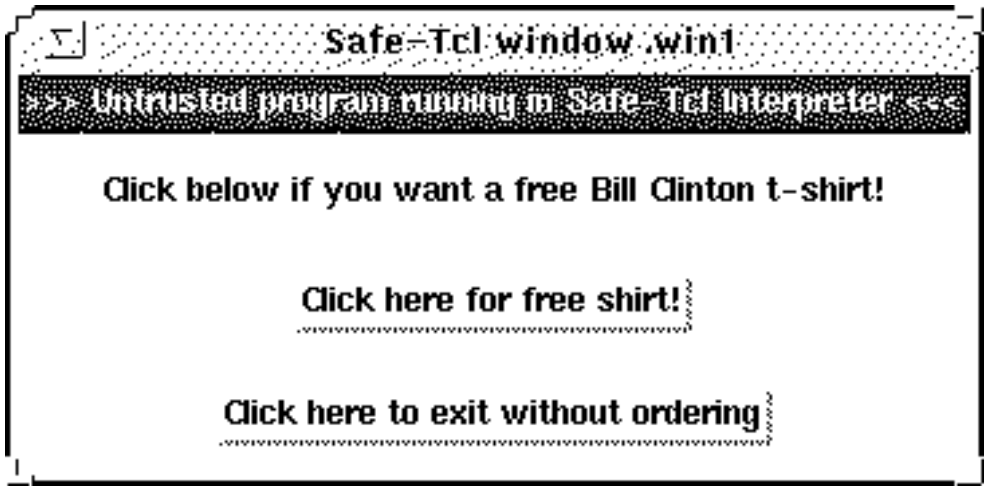
Although the Safe-Tcl interpreter does not currently implement any authentication mechanism, the language was designed with authentication in mind. In particular, the interpreter provides a global associative array variable, SafeTcl_services, which is used to indicate, among other things, the authenticated identity of the author of the program. If such an authenticated identity is available, then the value of SafeTcl_services(authentication) contains it, and this identity may be used by Safe-Tcl extensions to provide different levels of functionality to messages from trusted recipients.

IV. An Example Program

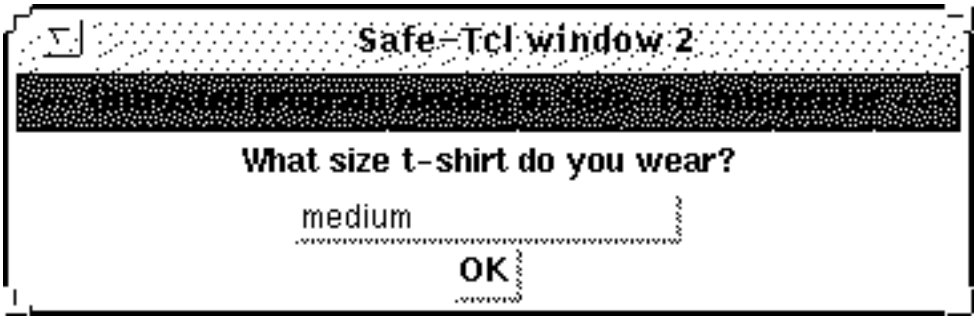
As with almost any programming language, a sample Safe-Tcl program long enough to be truly interesting would not fit in a paper such as this one. The following, however, is a simple program that demonstrates some important Safe-Tcl functionality. It offers the reader the opportunity to order a Bill Clinton T-shirt, using a graphical interface if Tk is running and a generic interface otherwise:

```
proc ordershirt {} {
  SafeTcl_sendmessage  -to tshirts@nowhere.really \
      -subject "Shirt request" \
      -body [SafeTcl_makebody "text/plain" \
              [SafeTcl_getline "What size t-shirt do you wear?"
                               "medium"] "" ]
    exit
}
if {[lsearch $SafeTcl_InterfaceStyle Tk3.*] >= 0} {
    set foo [mkwindow]
    message $foo.m -aspect 1000 \
      -text "Click below if you want a free Bill Clinton t-shirt!"
    button $foo.b -text "Click here for free shirt!" \
        -command {ordershirt}
    button $foo.b2 -text "Click here to exit without ordering" \
         -command exit
    pack append $foo $foo.m {pady 20} $foo.b {pady 20} $foo.b2 {pady 20}
} else {
    set ans [string index \
                [SafeTcl_getline "Do you want a free Clinton t-shirt? "
                                 "No"] 0]
    if {$ans == "y" || $ans == "Y"} {ordershirt}
    exit
}
```

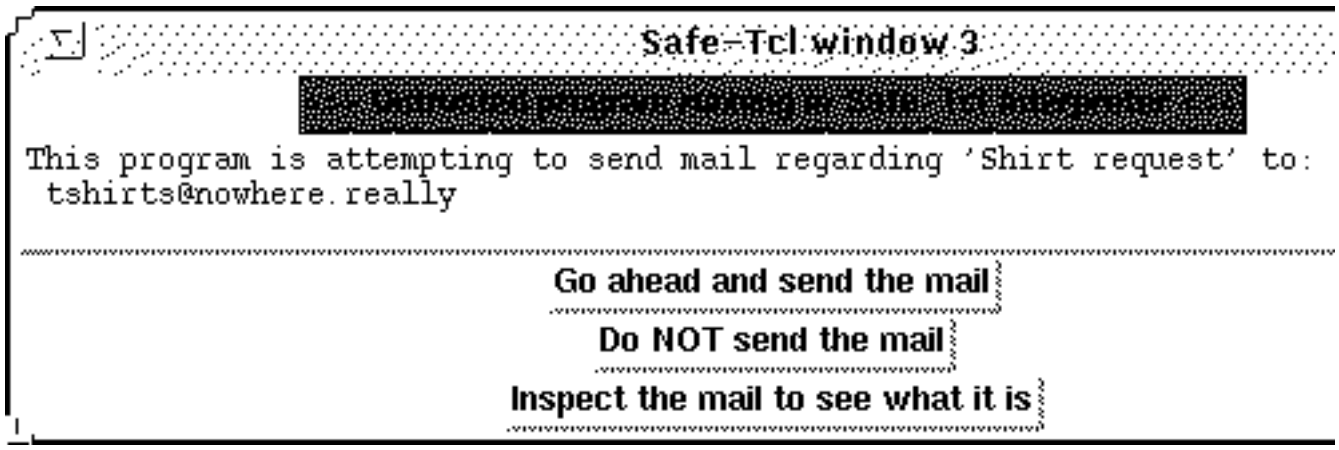
The screen images that follow show how this process proceeds when the Tk interface is in fact available.



The second screen:



The third screen:



V. Status, Availability, and Future Prospects of Safe-Tcl and Enabled Mail

The Safe-Tcl specification and our implementation of it were made freely available on the Internet in December, 1993. With the exception of the authentication mechanisms, the implementation completely implements the specification. The documentation and implementation may be obtained on the Internet via anonymous ftp from ftp.ics.uci.edu, in the file "mrose/safe-tcl/safe-tcl.tar.Z". An Internet Safe-Tcl mailing list may be joined by sending mail to safe-tcl-request@cs.utk.edu.

Unlike prior languages for enabled mail, the authors believe that the Safe-Tcl technology is complete and robust enough to serve as the first real open platform for mail enabled applications. It is our hope that many people in the Internet community will develop Safe-Tcl applications and extensions.

If successful, this technology will be, for most of the world, the first clear glimpse of the power of enabled mail. As such, we expect that it will teach us a lot about the design requirements for an enabled mail language. For that reason, we are not rushing to try to obtain any kind of official standardization status for the language, but will consider taking that step after accumulating substantial real-world experience with Safe-Tcl. Eventually, we hope that Safe-Tcl will either evolve into the standard language for enabled mail or else strongly influence the design of a successor language that will fill that role.

Acknowledgements

Safe-Tcl was implemented by Marshall Rose and Nathaniel Borenstein. The implementation would not have been possible, however, without the solid Tcl/Tk code base and consistently helpful advice provided by John Ousterhout. We have also benefited from the comments of several people in the process of designing Safe-Tcl, notably Dave Crocker, Ned Freed, Karl Lehenbauer, Daniel Newman, Rich Salz, Allan Shepherd, and Peter Svanberg. I am grateful to Ralph Hill and Marshall Rose for their comments on an earlier draft of this paper.

References

1. Borenstein, N; Thyberg, C. "Power, Ease of Use, and Cooperative Work in a Practical Multimedia Message System", Int. J. Man-Machine Studies, April, 1991.

2. Forsdick, H.C., et al., "Initial Experience with Multimedia Documents in Diamond", Computer Message Service, Proceedings IFIP 6.5 Working Conference, IFIP, 1984.

3. Borenstein, N., Freed, N. MIME (Multipuprpose Internet Mail Extensions), RFC 1521.

4.Ousterhout, J., "Tcl: An Embeddable Command Language.", Proc. USENIX Winter Conference, January 1990, pp. 133-146.

5. Ousterhout, J., *Tcl and the Tk Toolkit*, Addison-Wesley, Reading Massachusetts, 1994.

6. Ousterhout, J., "An X11 Toolkit Based on the Tcl Language.", Proc. USENIX Winter Conference, January 1991, pp. 105-115.

7. Goldberg, et al, "Active Mail - A Framework for Implementing Groupware", CSCW ´92 proceedings, Toronto.

8. Vittal, John, "Active Message Processing: Messages as Messengers", in *Computer Message Systems*, R. P. Uhlig, editor, North-Holland Publishing Company, 1981.

9. Hogg, John, "Intelligent Message Systems", in *Office Automation*, Dionysios Tsichritzis, editor, Springer-Verlag, 1985.

10. Borenstein, N. "Computational Mail as Network Infrastructure for Computer-Supported Cooperative Work", CSCW '92 proceedings, Toronto.

11. Linn, J., "Privacy Enhancement for Internet Electronic Mail: Part I - Message Encryption and Authentication Procedures", RFC 1421, IAB IRTF PSRG, IETF PEM WG, February 1993.

12. Zimmerman, Phil, "PGP Users's Guide", online documentation, June, 1993.

13. Rose, Marshall, and Nathaniel Borenstein, "A Model for Enabled Mail (EM)", draft in preparation.

14. Rose, Marshall, and Nathaniel Borenstein, "MIME Extensions for Mail-Enabled Applications: Application/Safe-Tcl and Multipart/enabled-mail", draft.

15. Coded Character Set--7-Bit American Standard Code for Information Interchange, ANSI X3.4-1986.