The Suitability of Tcl/Tk for Remote System Management

Dan Razzell
Starfish Systems
dan@starfishsystems.ca

Abstract

Starfish is a *managed agent* method for securely administering groups of computer systems. Because of the privilege necessary for the effective use of such a tool, its design demands careful attention to issues of security and clarity. It might not seem that an open, extensible scripting language such as Tcl/Tk would be suitable for such rigorous use, but on the contrary it has proven to be a nearly ideal choice of implementation language.

In this paper we will examine the general conditions for secure system management, and also look at some of the specific language features of Tcl/Tk which lend themselves to this class of activity. Throughout, we will try to convey some sense of the way in which open system architectures promote security and integration even, or especially, in heterogenous environments.

1. Introduction

Starfish is a method for securely administering groups of computer systems. It can be described in terms of three principal components: a *secure channel* for passing expressions and results remotely, lightweight *agents* which evaluate those expressions on remote systems, and a *manager*, placed somewhere on the network, which issues expressions to groups of agents and acts on their results. The agents and the manager are implemented as communicating Tcl interpreters. Each therefore supports a complete programming environment, and so each can be readily extended to meet site requirements. Integration is seamless because Tcl is used not only for implementation but also as the language for expressing remote control.[1]

2. Background

Computer systems which interact over a network behave as a *virtual machine* [45] having its own significant properties. The systems themselves may range from sharing no organizational relationship at all (for example, surfing the web) to being elaborately organized and managed under a common policy (for example, industrial process control.) Furthermore, the capabilities of these systems, and their degree of coupling, vary with changing requirements. The systems tend to develop increasingly complex patterns of affinity which are dynamic, overlapping, and not necessarily well behaved.

Any useful model of system management consequently has to account for a diverse range of virtual machines, for evolving capabilities, and for unforeseen behaviors. System management as a discipline is primarily concerned with how all of this diversity can be functionally integrated and made secure. In practice, it provides the expertise for infrastructure design and strategic planning. as well as conducting the ongoing transformation of the component systems which make up a given virtual machine.

In terms of transformation, its techniques fall into two broad categories according to the phase at which they are applied: those used to initially *install* systems and those used to *maintain* them. The distinction arises because each type of operating system conventionally (and rather jealously) provides a specialized installation mechanism [20][41][42] which is unsuited to other systems [2]. As we have little influence over these mechanisms, they have no further place in our present discussion. Instead, our primary focus will be the maintenance phase, where we are free to develop our own policies and mechanisms, and to apply them to our own heterogenous groups of systems.

A number of different techniques for incremental maintenance are commonly encountered in system management. At one end of the spectrum, systems may be individually managed at their consoles, or in a minor variation, may be remotely managed as if at their consoles [19][22][50]. Such techniques must be considered among the most primitive forms of system management, as they provide no abstraction in support of the virtual machine. Despite evident shortcomings, they remain extremely prevalent. At the other end of the spectrum are fully automated techniques for installing and maintaining systems [2][6][14][38]. These work well on a large scale and in highly ordered environments. They represent an ideal of system management which is conceptually attractive but often difficult to realize in practice. This is because computing environments are most often observed to evolve chaotically.[2] Few organizations, in practice, seem prepared to develop a detailed system management strategy and then follow it through to the point where fully automated techniques could be applied.

Techniques at both ends of the spectrum thus prove to be unsatisfactory in most environments. Consequently, somewhere in the middle have arisen various *ad hoc* techniques which offer some practical form of scalability along useful, but often specialized and therefore constrained, dimensions [8][28][34]. Starfish is an example of this intermediate class, oriented toward *ac hoc* management of groups of systems, but unusual in having a very general focus with no inherent constraints on capability. Its capabilities are powerful enough and secure enough to manage systems in chaotic environments, thus preparing them for a gradual migration toward our ideal of fully automated management. Starfish is implemented in open source, and uses open standards.

1 The Lisp Machine [18] was a more ambitious effort to unify the interactive environment with the implementation language, reaching all the way to the level of processor instruction set.

2 "The literature and common knowledge implies that there is some strong definition of how a machine should be configured in an environment. Yet the majority of the sites that were interviewed *could not say with certainty whether or not any of their hosts matched that definition;* they could only say that they were working without complaint." [11]

3. Design Goals

In order to be valuable to its community, Starfish needs to be:

- secure, open
- small, simple, easy to use and verify
- powerful, scalable, extensible, portable
- equally suited to interaction and automation

It might seem as though these criteria together constitute an overconstrained problem. However, we believe that Starfish demonstrates, first of all, that this particular problem space is in some sense actually synergistic, and second, that attractive solutions are entirely achievable with careful design.

4. Design Requirements

Our first essential design requirement for remote system management is simply to have a notation for expressing system management operations. As we will see later, any such notation must meet specific tests of *expressiveness*, *power*, and *portability:*

- It must be *expressive* in the computational sense of being suited to representing arbitrary algorithms for efficient execution. In contrast, other management tools may use notations which have limited expressive power or indeed are purely parametric [2][6][27]. Such limited notations can be valuable when managing ordered environments under predetermined conditions, but they belong at the idealized end of the system management spectrum. They would be unable to adapt to the novel and chaotic environments typically encountered by tools such as Starfish.

- The notation must be *powerful* in the sense of being able to reach, without constraint, into every part of the system being managed. How well this can take place depends in practice on the capabilities of each individual system, and to some extent on the architectural commonality among systems being managed. We may decide to limit these powers under certain conditions, but such limits should be imposed as a policy decision, not a design decision.

- Finally, the notation must be *portable* not only in the sense of providing graceful support for existing commonality, but also in its ability to form new abstractions for common integrative purposes.

Our second essential design requirement relates specifically to the remote aspects of system management, and states that communication between interacting systems must be *architecturally secure*. Secure remote management calls for secure communication, whose methods are traditionally analyzed in terms of *integrity*, *confidentiality* and *authentication* [10]. Though all three factors are clearly relevant to system management, authentication turns out to be especially interesting because of the high level of privilege which must be exercised during typical system management tasks. Under these conditions, a group of communicating systems forms a privileged metasystem in which the security of one system might have consequences for the others. Our design must therefore not only authenticate to prevent outside interference but also to make explicit the transfer of privilege [17][21][35] within the metasystem.[3]

5. On Security and System Architecture

It would be irresponsible to discuss security without some reference to architecture. Unlike many other system concepts, security is not a component that can be meaningfully added or removed from some isolated part of an architecture. Security is only meaningful when considered as a property of the entire design. For example, the security of a building is not much enhanced by installing elaborate locks, as long as holes can be kicked through the walls, or hinge pins removed. Security must be considered not only in *every element* of the architecture, but in every *relationship* between elements as well.[4]

Security is therefore better understood as an envelope which must enclose every accessible surface of a system [31][33]. Only if the envelope is complete and intact can the system be considered architecturally secure. In a distributed metasystem, this envelope must enclose all of the component systems as well as the network which connects them. Then we can reason about security simply by examining the properties of a particular envelope.

In practice, a model this simple is very difficult to realize. Real metasystems have complex, overlapping relationships with other metasystems, and may involve elements, such as networks, which are not themselves secure. There is still hope for security, however, as long as we are able to put a complete envelope around some given subsystem. If we can do this for a number of subsystems, a further step would be to metaphorically cut a hole in each envelope, and then construct a connecting tunnel between the holes. We now have a single, topologically complete envelope consisting of the original envelopes, the holes, the tunnel, and the joints between the tunnel and the holes.

In this considerably more complex model, our ability to reason about security will thus depend on our total knowledge of the properties of each of these elements, as well as their topology and relationships. Ironically, therefore, the security of the operating system element is an essential *prerequisite* for secure system management [49], even as it is the intended *target* for system management tasks aimed at improving security.

To conclude this brief examination of secure architecture, we can observe that security favors simple, structurally consistent metasystems, built using a small number of clearly defined elements, all having known properties and relationships. It follows that when Starfish, or any other system management tool, becomes a component of such an architecture, its role and its properties should likewise be simple and clearly defined. These will become important factors to bear in mind when we consider the suitability of an implementation language.

---

3 Such insights are not always immediately obvious. It was not until about 1988 that the Unix community realized that whereas ordinary user privileges could safely be made transitive across systems, superuser privileges could not.

4 "Cryptography differs from all other fields of endeavor in the ease with which its requirements may *appear* to be satisfied." [10]

6. On Security and Remote Management

As with the preceding discussion of architecture, we are obliged to briefly turn our attention to consider how remote system management contributes to security. In other words, what motivates the use of a remote system management tool in the first place? Although we will use a similar kind of security language, note that this will be something of a departure from our main question of what it entails to build such a tool.

Consider the proposition that *consistency* is the primary condition for system security. Even in a single isolated system, we need some way to enforce consistency between our model of the system and its physical implementation. We observe that the security of a system depends on how it has been *configured*, for example what software has been installed and how each of its features have been chosen [11][13][38]. Therefore, unless there is some external, independent way to verify critical aspects of its configuration, we have no basis for claiming that a system is secure. In a metasystem, as we have seen, the nature of the security envelope calls for further consistency among each of the component systems, since otherwise any weakness in one component will compromise the others. Moreover, the configuration space and the potential for unforeseen interactions are both simply larger in a metasystem than in any of its components.

Consistency is likewise the primary condition for system integration in general. The situation is comparable to the need for version control [43] within a software development project.[5] Internally consistent groups of systems have predictable behaviors and interoperate cleanly. Conversely, gratuitous inconsistencies between systems tend to compound over time, producing divergent behaviors which become harder to correct, or even to diagnose, as dependencies form. Indeed, whenever we encounter computing environments in crisis, we invariably find that they suffer from poor regard for consistency [12].

Whether for the purposes of integration or security, system management thus has as its core activity the enforcement of consistency in an environment which would otherwise tend, under many influences, toward chaos. Effective system and network management is a natural companion to effective system and network security, to such an extent that it seems appropriate to treat them as a single methodology.

7. Language Considerations

With this background, we can now return to the central topic of language suitability. How do we decide in general whether a language is suitable for remote control of an operating system? The answer depends on the convergence of two sets of factors, one having to do with the expressive characteristics of the language itself, and the other with the capabilities of the operating system and its methods for controlling them.

7.1. Expressiveness

Apart from structural features such as regularity and composability by which one programming language can be abstractly compared to another, some language characteristics are especially suited to the practical realities of system management. For example, often even the most *ad hoc* inspection and adjustment of system state involves repetitive or recursive actions such as renaming a set of files or traversing a directory tree. While a variety of command shell languages [4] have specialized features for performing these actions, they often come at the expense of linguistic completeness, for example with respect to composability. Remarkably few shell languages fully meet our test of expressiveness, and many exhibit curious pathologies of syntax[6] which interfere with good programming practices.

Interpreted programming languages such as Lisp [26], Tcl [32], and Forth [36] have the desired structural coherency while preserving the important ability to express simple operations in a simple way. Indeed, all three have been used with varying success as command shell languages, but of these, Tcl seems to have struck the best compromise in recognizing, along with traditional shell languages, that a line of text is a natural unit of computation equivalent to a function call. By comparison, Lisp requires such expressions to be delineated with parentheses, while Forth has no delineation at all. The former has been found to be a practical annoyance, while the latter leads inevitably to confusion.

Rather surprisingly, the strongest *linguistic* argument in favor of Tcl for system management proves to be its simple ability to integrate with native command shell syntax, and thus provide access to native system services without recourse to special quoting or escaping mechanisms. In this critical characteristic[7], it contrasts with alternative scripting languages such as Python [25] and Perl [47], to say nothing of languages such as Java [16] which must provide completely different abstractions in place of native services.

7.2. Power

The question of language suitability is not as easily bounded where factors of the operating system at large are concerned. Consider that any given operating system may provide arbitrarily many capabilities, using any conceivable form of interface. Not all of these may be equally well devised for remote or programmatic control. Indeed, with extraordinarily few exceptions, operating systems make no particular commitment to provide the same capabilities in the same way to an interpreted command environment as they do to application programs.

However, after some forty years of operating system development we can at least make some practical observations. The first is that most operating systems have in fact developed means to control their services using a command line interface, notwithstanding the various ways in which these services may

5 While version control involves organizing multiple configurations of a single system, integration imposes a single configuration on multiple systems.

6 "The conventions are so simple and regular that they are trivial to learn. By comparison with most other languages, Tcl syntax is a pleasure to use." [24]

7 "This is a very important feature because this type of remote access is the lowest common denominator that can usually be found on almost any host or network device in the field. It requires neither a graphical interface nor the installation of special software." [28]

be implemented internally. The second is that the Tcl community has provided so many interfaces to diverse programming environments that for practical purposes we may be satisfied that if a system service is available programmatically, it can be made available in a command line model. Indeed, all of Tcl extensibility is based on this premise.

There are limits to such an assumption, of course, since much depends on the commitment by the operating system to principled design. System facilities which can be controlled, for example, only over a GUI, or only from the console, are limited by their design in a way that no language can overcome.

7.3. Portability

Ordinarily, we think of software as being portable when it can function in a way that is uncompromised by its environment. The core Tcl/Tk interpreter is able to achieve portability in this fundamental and necessary sense as much through careful implementation[8] as by the use of language formalism.

Although perhaps sufficient for application programming, this sense alone is unfortunately *not* sufficient where system programming is concerned. Operating systems differ precisely because they provide differing capabilities and abstractions. We argue that a mechanism for remote system management which *only* supports a reduced set of features generic to all operating systems can likewise have only limited practical application.

Instead, for system programming the ordinary sense of portability has to be somehow revised so as to (selectively) extend access to distinctive features of the system environment, as well as to create unifying abstractions for features which differ across systems. In this respect, it proves to be more important that an implementation language provide flexibility for abstraction, than for it to impose a rigid model of abstraction, however elegant that model might seem.[9] We have enough to do just to cope with the artifacts of different operating system designs, without compounding the problem with artifacts of our own.

The careful pragmatism by which Tcl/Tk achieves ordinary portability works to its advantage under this special condition of *systemic portability*. Not merely can the interpreter be extended to access arbitrary system features, but just as importantly, the Tcl/Tk development community has shown consistent insight and diligence in designing these extensions with portability in mind. This last characteristic may not be part of the language proper, but it undeniably influences its suitability for system programming projects such as Starfish.

8. Language Features in Tcl/Tk

Exotic language features are not the first choice when building secure systems. Besides the obvious concerns for program clarity and portability, the use of a smaller and more prosaic code base provides fewer potential sites of vulnerability, and encourages more frequent exposure of the code to peer scrutiny.

As Starfish demonstrates, the core language features of Tcl/Tk are naturally well suited to secure remote system management. The following list describes features which we feel are noteworthy:

- **Extensions and Channel Stacking**

  The `TLS` extension [44] provides stacking of SSL [9][15] security onto arbitrary Tcl channels. Therefore, neither Starfish nor Tcl have to implement this element of the security envelope directly, but instead can use the leverage available through existing software, thereby improving reliability.[10]

- **Slave Interpreters**

  The essential function of a Starfish agent is to evaluate expressions which it receives from a remote manager. In Tcl, this can be done elegantly (and almost unremarkably) within a slave interpreter, whereas most other languages would require some kind of dispatch table in order to parse expressions and perform the evaluation. The essential behavior can be captured in just a few lines:

```
set slave [interp create]
$slave alias unknown slaveunknown
set result [$slave eval $text]
```

- **Command Isomorphism**

  Most operating systems provide a command language interpreter whose basic line syntax closely resembles a Tcl function call. The respect in Tcl for these syntax conventions allows a Starfish agent to use the Tcl `unknown` function to gracefully transfer control from the interpreter to the operating system, or conversely to preempt that transfer in favor of some behavior internal to the agent.

- **Simple Event Model**

  It should come as no surprise that remote system management needs to operate asynchronously in order to scale well. The Tcl event mechanism provides a compact abstraction which proves entirely sufficient for asynchronous management. It is worth noting, however, that even this simple abstraction introduces significant complexity and consequent risk of confusion. The more elaborate thread parallelism would certainly *not* be a better choice for use in a security tool.

- **Simple Exception Handling**

  Exceptional conditions are commonly encountered in remote system management. We typically think of an expression as evaluating to some value, and by induction we can in turn understand compound expressions and predict the behavior of control structures. Expressions evaluated remotely,

8 "Note that language implementations tend to be written in themselves, particularly for their libraries. Perl's implementation is written mostly in Perl, and Python is written mostly in Python. Intriguingly, this is not true for Tcl." [48]

9 "Tcl tries very hard not to force a particular view of the world." [24]

10 Recent improvements to SNMP [7][23] may permit it to be used as an alternative transport. We are not aware of a Tcl extension which provides access to this new functionality.

however, may instead drop communication, hang indefinitely, time out, return no value, or return an infinite sequence of values. Depending on our knowledge of the remote system, such behavior many be expected or unexpected, and we may wish to have some control over how each case is handled.

Although it might seem that an elaborate exception mechanism would be needed to classify and handle the wide range of status codes, overflows, timeouts and so on which might arise, the overriding requirement for scalable remote management again turns out to be *simplicity*. We find it sufficient to tag all remote evaluations with a simple marker indicating success or failure, so that managers and agents can anticipate how to sequence through compound expressions which, in the nature of system management, sometimes yield unexpected results.

The `catch` and `error` functions have so far proven entirely sufficient for Starfish to model and propagate this basic behavior. Should the need for richer exception classing ever arise, the Tcl `errorCode` variable can be used to carry the additional information. We do not anticipate the need to extend control structure using the construct:

```
return -code n
```

- **Simple Callback Model**

  Callbacks are used as a technique for passing function bindings so that they can be invoked at some later time. A classic dilemma for language designers is how to reconcile the differences in scope between when the callback is bound and when it is invoked. The concept of a *closure*, although viable, introduces new and subtle considerations [1].

  Tcl resolves the scoping dilemma by invoking the callback at top level. This characteristically simple convention makes callbacks immediately attractive and useful. Indeed, Starfish would not be able to make effective use of OpenSSL [30] without access to the callbacks it provides for password management and certificate chain traversal.

- **Simple GUI**

  About 40% of the Starfish manager is related directly to the graphical user interface. This should be considered a comparatively *small* ratio, reminding us that much of the design of any application is influenced by its GUI. Our emphasis in Starfish, however, should not be on the GUI, but rather on the challenges of secure system management. It is therefore fortunate that Tk provides a simple, compact GUI whose defaults permit a minimalist style and encourage portability.

- **Soft Data Typing**

  Strong typing is useful for detecting type faults in environments whose structure can be rigidly defined. These are not, generally speaking, the environments in which system management takes place. Operating system components may well enforce strong typing for internal data, but the system as a whole rarely presents a unified type architecture that could be used by an external agent.

  Furthermore, the agent itself must be able to work with many different systems, where no common type architecture can be expected.

  For our purposes then, weak typing is vastly more appropriate. In practice, the additional burden of care on the developer proves to be minor, and the net result is simpler, clearer program code.

- **Soft Argument Lists**

  In a similar vein, soft argument lists have the effect of making Tcl applications simpler and more versatile. In addition, they prove to be compatible with command shell syntax, eliminating another potential source of complexity.

- **Simple Defaults**

  Although not strictly a property of the language, simple defaults are ubiquitous in Tcl because the language encourages them through soft data typing and soft argument lists. Good defaults lead to economy of expression, as well as to an inexpensive source of design clarity, as is evident in much of the Tcl development effort.

- **Binary Data**

  Starfish does not have a specific need to represent data in binary form. However, we were very relieved when this capability was added to Tcl, as the need for it could suddenly arise due to some unforeseen protocol or system consideration.

9. Comments on Scale and Style

It can be difficult to speak objectively about just what programming environment is suited to what scale of application. Subjectively, however, we can report that Tcl/Tk has consistently felt like the right fit throughout the Starfish project.

Tcl/Tk encourages, and in some sense demands, a style of incremental development in which a working prototype is allowed to take form quickly, then to be elaborated, revised, or even discarded at minimal cost. Though this style of development tends to expose good design in projects of modest size, it should not be expected to suit every project of every size. We can, however, report that, at least in the case of Starfish, it meets the design requirements for secure system management, an activity in which, we would argue, the standards are unusually rigorous. It may also be that the fit between language and application is especially apt in our case because system management itself is conducted as an ongoing and incremental process.

As a final point on the subject of appropriate style, we note that Tcl/Tk favors open source distribution. There are certainly areas where the suitability of open source could be debated, but for peer scrutiny of a cryptographically secure system there is no contest. Mechanisms *must* be exposed in order for security claims to be credible.

10. Comments on Security and Performance

The Starfish agent is defined completely in Tcl in about 850 lines of code. It takes one second to launch on a 100MHz Intel processor and occupies about 3 MB of virtual memory, which puts its resource demands on par, for example, with `syslogd`.

The core Starfish manager is defined in about 4800 lines of Tcl/Tk. It launches in three seconds on a 100MHz Intel processor and occupies about 5MB of memory, roughly half that of `xemacs`.

By comparison, a complete OpenSSL session handshake takes about one second using 384 bit RSA keys in both directions, and assuming zero network overhead. This should be considered an optimistic benchmark, as longer keys are ordinarily used. Public key crypto and key exchange are, of course, known to be computationally expensive [37].

These numbers serve to illustrate that the *total* application overhead of Starfish implemented in Tcl/Tk is comparable in speed to the *best case* overhead of a single asymmetric key exchange in native code.

A tight code base is always a valuable design goal, but never more so than in a security product. Less code not only means intrinsically fewer points of vulnerability in the product, but also more thorough exposure to peer scrutiny. The use of the OpenSSL library not only reduces the Starfish code base, but also takes advantage of the exposure and validation already contributed by products such as Apache [3] and OpenSSH [29]. Furthermore, it encourages such work to continue, to the benefit of *every* user of the digital commons.

In summary, secure remote system management is a domain which emphasizes simplicity and clarity, making modest performance demands beyond those associated with essential cryptography. From this perspective, system management tools are eminently suited to implementation in an interpreted scripting language such as Tcl/Tk in which the cryptosystem can be delivered in native code.

11. Conclusions

System management is an eminently *practical* field of application in which to examine programming language design. Expressiveness, power, portability, and security are all important indications of language suitability. In each of these, Tcl/Tk demonstrates outstanding strengths for system management.

The size and complexity of the Starfish project also proves to be a good fit for Tcl/Tk. Involving a few thousand lines of code and being oriented specifically toward peer scrutiny and site customization, it tries, like Tcl itself, not to force a particular view of the world.

We have further seen that the relationship between application and language becomes especially intimate in a project such as Starfish where the theme of integration is played out at many levels. It evidently remains an effective basis for comparison of system programming languages.

12. Future Directions

The foundational work for Starfish is complete. We have found that, at least where system management is concerned, small is beautiful [46]. What remains are some very important questions concerning the practical limits of systemic portability and extension. Will the system management community adopt Starfish and find its extensibility worth exploring in order to meet specialized site requirements? Given the variety of operating systems and the diversity of site requirements, how much convergence might then be possible toward a common set of system management primitives that all sites would find useful? Could this process in turn help to guide operating system design toward more unified architecture for secure system management? And not least, at exactly what point can *ad hoc* methods be handed over to automated methods?

13. Related Work

Starfish takes its inspiration from a simple login multiplexing tool named `octopus`, written by George Phillips at the University of British Columbia in 1989. Starfish was originally implemented as a modification to Scotty/Tkined [39][40], and after several years of experimental use was subsequently redesigned and rewritten directly in Tcl/Tk. We are deeply grateful for the assistance given by Jürgen Shönwälder during the Scotty/Tkined development phase, as well as to Eric Young and Tim Hudson for their work in bringing network cryptography within reach of the open source community in the form of OpenSSL.

Much interesting work on distributed system administration appears in the LISA conference proceedings, particularly in the years around 1994. It is remarkable how much of this work uses Tcl/Tk as the implementation language. Several papers explore concepts similar to Starfish, notably those by Pierce [34] and by DeSimone and Lombardi [8]. Unfortunately, these projects seem to have since lost momentum. Instead, emphasis has shifted to automated configuration methods, with some of the most consistent and principled work being reported by Burgess [5]. The difficulty with such methods, as we have noted, is that many environments appear too chaotic to take advantage of them. Starfish exists to help turn that situation around.

14. Availability

Starfish is available under the GNU General Public License from **www.starfishsystems.ca**.

References

[1] J. Allen, *Anatomy of LISP*, McGraw−Hill, 1978

[2] P. Anderson, "Towards a High−Level Machine Configuration System," *Proc. LISA 1994* (Sep 1994)

[3] Apache, `http://www.apache.org/`

[4] S. Bourne, "The UNIX Shell," *Bell System Technical Journal*, vol. 57, no. 6 (Jul 1978)

[5] M. Burgess, "Computer Immunology," *Proc. LISA 1998* (Dec 1998)

[6] M. Burgess, R. Ralston, "Distributed Resource Administration Using Cfengine," *Software − Practice and Experience*, vol. 27, no. 9 (Sep 1997)

[7] J. Case, R. Mundy, D. Partain, B. Stewart, *Introduction to Version 3 of the Internet−Standard Network Management Framework*, RFC 2570, The Internet Society (Apr 1999)
[8] S. DeSimone, C. Lombardi, "Sysctl: A Distributed System Control Package," *Proc. LISA 1993* (Nov 1993)
[9] T. Dierks, C. Allen, *The TLS Protocol Version 1.0*, RFC 2246, The Internet Society (Jan 1999)
[10] W. Diffie, M. Hellman, "New Directions in Cryptography," *IEEE Trans. Inform. Theory*, vol. IT−22, no. 6 (Nov 1976)
[11] R. Evard, "An Analysis of UNIX System Configuration," *Proc. LISA 1997* (Oct 1997)
[12] R. Evard, "Tenwen: The Re−Engineering Of A Computing Environment," *Proc. LISA 1994* (Sep 1994)
[13] J. Finke, "Automation of Site Configuration Management," *Proc. LISA 1997* (Oct 1997)
[14] R. Finkel, "Pulsar: An Extensible Tool for Monitoring Large Unix Sites," *Software − Practice and Experience*, vol. 27, no. 10 (Oct 1997)
[15] A. Frier, P. Karlton, P. Kocher, *The SSL Protocol Version 3.0*, Internet Draft, Netscape Communications (Mar 1996)
[16] J. Gosling, B. Joy, G. Steele, G. Bracha, *The Java Language Specification Second Edition*, Addison−Wesley, 2000
[17] R. Graham, "Protection in an Information Processing Utility," CACM , vol. 11, no. 5 (May 1968)
[18] R. Greenblatt, T. Knight Jr., J. Holloway, D. Moon, D. Weinreb, "The LISP Machine," *Interactive Programming Environments*, McGraw−Hill, 1984
[19] Hewlett−Packard, *Managing Systems and Workgroups: A Guide for HP−UX System Administrators, B2355−90742 (Jun 2001)*
[20] Hewlett−Packard, *Ignite−UX Administration Guide*, B2355−90749 (Mar 2002)
[21] R. Housley, W. Ford, W. Polk, D. Solo, *Internet X.509 Public Key Infrastructure: Certificate and CRL Profile*, RFC 2459, The Internet Society (Jan 1999)
[22] IBM, *System Management Interface Tool*, (Nov 2000)
[23] J. Levi, J. Shönwälder, *Definitions of Managed Objects for the Delegation of Management Scripts*, RFC 3165, The Internet Society (Aug 2001)
[24] D. Libes, "Writing a Tcl Extension in only 7 Years," *Proc. Fifth Annual Tcl/Tk Workshop* (Jul 1997)
[25] M. Lutz, *Programming Python*, O'Reilly, 2001
[26] J. McCarthy, P. Abrahams, D. Edwards, T. Hart, M. Levin, *LISP 1.5 Programmer's Manual*, MIT Press, 1962
[27] T. Miller, C. Stirlen, E. Nemeth, "satool: A System Administrator's Cockpit, An Implementation," *Proc. LISA 1993* (Nov 1993)
[28] E. Mitchell, E. Nelson, D. Hess, "ND: A Comprehensive Network Administration and Analysis Tool," *Proc. LISA 2000* (Dec 2000)
[29] OpenSSH, `http://www.openssh.org/`
[30] OpenSSL, `http://www.openssl.org/`
[31] E. Organick, *The MULTICS System: An Examination of Its Structure*, MIT Press, 1972
[32] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison−Wesley, 1994.
[33] D. Peterson, M. Bishop, R. Pandey, "A Flexible Containment Mechanism for Executing Untrusted Code," P*roc. Eleventh USENIX Security Symposium* (Aug 2002)
[34] C. Pierce, "The Igor System Administration Tool," *Proc. LISA 1996* (Sep 1996)
[35] K. Ramm, M. Grubb, "Exu − A System for Secure Delegation of Authority on an Insecure Network," *Proc. LISA 1995* (Sep 1995)
[36] D. Rather, D. Colburn, C. Moore, "The Evolution of Forth," *ACM SIGPLAN Notices*, vol. 28, no. 3 (Mar 1993)
[37] R. Rivest, A. Shamir, L. Adelman, "A Method for Obtaining Digital Signatures and Public−Key Cryptosystems," *CACM*, vol. 21, no. 2 (Feb 1978)
[38] J. Rouillard, R. Martin, "Config: A Mechanism for Installing and Tracking System Configurations," *Proc. LISA 1994* (Sep 1994)
[39] J. Shönwälder, H. Langendörfer, "Tcl Extensions for Network Management Applications," *Proc. Third Annual Tcl/Tk Workshop* (Jul 1995)
[40] J. Shönwälder, H. Langendörfer, "How to Keep Track of Your Network Configuration," *Proc. LISA 1993* (Nov 1993)
[41] Sun Microsystems, *Solaris 9 Installation Guide*, 806−5205 (May 2002)
[42] Symantec, *Symantec Ghost Implementation Guide*, 07−30−00482 (Nov 2001)
[43] W. Tichy, "RCS − A System for Version Control," *Software − Practice and Experience*, vol. 15, no. 7 (Jul 1985)
[44] TLS extension for Tcl, `http://sourceforge.net/projects/tls/`
[45] S. Traugott, J. Huddleston, "Bootstrapping an Infrastructure," *Proc. LISA 1998* (Dec 1998)
[46] E. Schumacher, *Small is Beautiful*, Harper & Row, 1973
[47] L. Wall, T. Christiansen, J. Orwant, *Programming Perl*, O'Reilly, 2000
[48] D. Wheeler, *More Than a Gigabuck: Estimating GNU/Linux's Size*, `http://www.dwheeler.com/sloc/`
[49] C. Wright, C. Cowan, G. Kroah−Hartman, J. Morris, S. Smalley, "Linux Security Modules: General Security Support for the Linux Kernel," P*roc. Eleventh USENIX Security Symposium* (Aug 2002)
[50] T. Ylönen, "SSH − Secure Login Connections over the Internet," *Proc. Sixth USENIX Security Symposium* (Jul 1996)