The following paper was originally published
in the Proceedings of the
Tenth USENIX System Administration Conference (LISA X)
Chicago, IL, USA, Sept. 29 - Oct. 4, 1996

Shuse: Multi-Host Account Administratio

Henry Spencer
SP Systems

Shuse: Multi-Host Account Administration

Henry Spencer – SP Systems

ABSTRACT

At the beginning of 1995, Sheridan College urgently needed an organized way of administering a large number of user accounts spread across multiple Unix systems. With 6000+ accounts on a network that had recently undergone dramatic and ill-coordinated growth, the situation was already nearly unmanageable; with the user population forecast to double in autumn, disaster loomed. NIS served reasonably well for the simple task of distributing password files, but maintaining the master copy was proving problematic, creating directories and configuration files for new users was a very ad-hoc process, and there was no obvious place to record assorted supplementary information.

The response was to create a new software package, dubbed "Shuse" for "Sheridan user management". A central daemon maintains the user database, which is in a fully extensible text-based format. Rather than use a commercial database package, the daemon simply keeps the entire database in its (virtual) memory, and the master copy on disk is optimized for rapid updates rather than efficient access. (RAM is cheaper than database packages nowadays.) Update requests go to the central daemon; it invokes auxiliary processes on other hosts as necessary to create, destroy, and move user files.

Shuse is written essentially entirely in Expect, an extended variant of Tcl. Inter-host communication is done by using Expect's process-control primitives to fire up *telnet* processes; bulk data transfer is done via NFS. About 100 lines of C code, in three small auxiliary programs, provide services that are not present in Expect. A not-accidental byproduct of this approach is near-automatic portability and correct functioning even in a heterogeneous network.

Shuse is in operational use, currently administering over 20,000 user accounts (the forecasts were low). Various problems were encountered along the way, some easily solved and some requiring considerable unforeseen effort. The use of Expect has been a clear success, performance problems were easily resolved, and the central-daemon approach has worked well.

The Problem

Sheridan College[1] is a large community college, with several campuses located in the outer suburbs of Toronto. Its computing facilities are centered on a set of DEC Alphas running DEC UNIX (formerly named OSF/1), although there are also large numbers of PCs, a scattering of high-end graphics machines for the animation courses, and a wide variety of odds and ends (everything from 486/Pentium boxes running BSD/OS to one or two moldering VMS machines). The facilities have expanded enormously in the last few years, and there has been only limited advance planning on how to deal with the rapid growth.

Sheridan has been moving steadily toward an accounts-for-everyone policy, exacerbating the usual difficulties of large numbers of accounts on multiple hosts. In early 1995 there were over 6000 users; this was forecast to double within the year. Many of the users have very little experience with computing, especially shared multi-user computing, and the combination of heavy course-loads, non-technical backgrounds, and high turnover limits what can be done with user education. (Instructions like "change your password by doing an *rlogin* to server so-and-so and running *yppasswd*" are worse than useless.)

By the beginning of 1995, rapid growth and limited planning had made the situation almost intolerable. Funding shortages prevented major increases in staff, routine maintenance chores like creation of new accounts were absorbing large amounts of staff time (to the point where software problems were not getting solved because nobody had time), and continued growth threatened total collapse. Improvements were urgently needed, and in particular, really had to be in place before the September 1995 onslaught. At this point, I was brought in to Do Something About It.

Some potential difficulties were *not* present. All hosts do share a common password file, partly because they share file systems quite extensively via NFS. While there is a lot of heterogeneity "around the edges", the major servers are all the same type of machine running the same operating system. (This

[1] The one area where LISA attendees might perhaps have heard of Sheridan is that its computer-animation program has an international reputation.

didn't help as much as one might think, however, because it was not clear that this simple situation would persist.) The number of hosts involved as major participants is quite small: the primary problem was large numbers of users, not large numbers of computers. Finally, the network and the major servers have been reasonably reliable, and it was decided that there is no requirement to preserve full functionality in the presence of dead servers or partitioned networks.

Existing Software

An admittedly rather cursory look at existing solutions to this problem revealed little that seemed helpful.

Some system suppliers offer proprietary account-management software for networks of their machines, but Sheridan's network was already slightly heterogeneous and might easily become more so, so a system-specific approach was unattractive. The supplier packages also have an annoying habit of being menu-driven GUI-based interactive programs, which may be easy to use when creating a single account, but are severely unsuited to environments where 5000 accounts must be created in a week or two. Besides, DEC would be the only reasonable supplier for such a thing in this case, and DEC didn't appear to offer anything suitable.

The MIT Athena project has a Service Management System [1] which addresses this problem. (Indeed, it is somewhat similar to what we eventually built.) Unfortunately, it relies on a commercial database package *and* on other Athena software, and this didn't look like it was going to drop easily into Sheridan's existing environment.

At the time, we were not aware of GeNUAdmin [2] or AGUS [3], which might perhaps have been suitable.

Finally, one thing that was very clear was that Sheridan wanted a definitive solution to the problem, not a temporary bandage for the wound. This precluded various ad-hoc solutions which might have postponed the crisis at the cost of more effort later.

Design

Although some complications have been added to the original design, the basic elements have been fairly stable.

The fundamental approach was largely determined by consideration of one issue: coordinating updates. The orthodox way to do this, in a shared system, is some kind of locking protocol... but that presents problems in an NFS environment. None of the usual Unix file-based locking techniques work reliably with NFS's shoddy imitation of Unix file semantics. Reliable locking in such an environment requires using a supplementary protocol to consult a daemon somewhere; this is the approach taken by NFS's own locking primitives, but unfortunately they are notoriously buggy.

Given that we were going to have to implement our own daemon anyway, the obvious approach was just to have it do all the work. Locking is unnecessary when all requests are funneled through a single ''secretary'' process. A dedicated central-information-server system was available to serve as the daemon's host, and its reliability and uptime were sufficiently good that the extra complications of distributed redundancy could be avoided. So we decided to implement a single central daemon, which would respond to queries, perform database updates, and invoke subordinates on other servers as necessary.

With this approach, synchronization is a non-issue, since only one process ever modifies the database. At least for starters, we decided to avoid re-introducing concurrency via threads: there is a single stream of control, operating in an event-based loop. The payoffs are a complete absence of locking overhead, fast updates, and vastly easier debugging (that last being particularly attractive in view of the hard deadline).

This approach was even more attractive because it permits a very useful optimization: the central daemon can be left running permanently, and can simply cache the entire database in its memory. One might think that this technique would be suitable only for small databases, but done well, it scales up quite effectively – memory is cheap. In any case, a brief analysis indicated that the amount of information for a particular user was unlikely to exceed a few hundred bytes, and this would require only a few megabytes for the expected user population. (More importantly, some quick tests showed that nothing dire would happen if this prediction was significantly exceeded.)

Obviously, it is still necessary to have an on-disk copy of the database, so that updates will survive both planned downtime and crashes. With all read-only accesses satisfied from the daemon's memory, the on-disk copy can be optimized for cheap and simple updates rather than rapid access to large amounts of data. After some quick feasibility testing, we decided to simply make each user's data a separate file in a simple text format. This does make the daemon's startup rather slow, since it has to read thousands of tiny files, but with a permanently-running daemon this is not needed often. Experiments indicated that on an otherwise-quiet system, reading 10-20,000 small files took only a few minutes, which seemed tolerable for a relatively rare event. We considered subdividing the files into a directory tree, but experiments indicated that just keeping them all in one directory was quite workable on a modern system.

For the file format itself, we briefly considered various extended versions of the classical *passwd*-file format, but decided against it. Any format with a fixed number of fields suffers when requirements change, as witness all the creative things that have been done with the "GCOS" field in the *passwd* file. While it would be necessary to generate a *passwd* file from the database, we wanted the database itself to be flexible and extensible, so it could contain *all* the information about a user and would not need supplementing with auxiliary databases as new needs appeared. We did opt for a text-based format, partly just for simplicity, partly because this makes a wide variety of Unix tools useful for setting up the database or doing emergency surgery on it. A user's database file looks something like Figure 1. The *server* field contains a code identifying the system the user's home directory resides on; there is a separate control file which maps server codes to host names, to simplify changes in host configuration. The *schema* field contains a code indicating how to build an initial home directory for a new user; it is passed as an argument to the script that actually builds the directory. Fields like *office* and *workphone* contain information that is assembled into a suitable "GCOS" field for the *passwd* file; since we want the Shuse database to be the primary database, not a derived one, we store the information broken down by meaning, instead of a preformatted version appropriate to one specific version of the *passwd* file.

A single centralized daemon could not do the entire job. In particular, when creating or deleting users, it would be necessary to operate as *root* on the host holding the user's home directory, and the limitations of NFS required that to be done locally. This was also necessary for a different reason: Sheridan mounts only subsets of its filesystems on its individual servers, so the central server host cannot necessarily see the filesystem which would have to be updated. It seemed that it would be necessary for the daemon to invoke an auxiliary program on the other, "slave", servers. (Having this done as needed by the daemon, rather than at regular intervals by *cron*, would propagate updates more quickly and avoid unnecessary overhead.)

We decided that the auxiliary program would read a description of what users *should* be on its host, then examine the host to find out which users were actually present, and then act to correct any discrepancies. This seemed likely to produce more robust operation than having the daemon send update commands to the auxiliary.

At this point we were starting to need names for the programs. We dubbed the whole system "Shuse" (pronounced like "shoes"), the daemon "shused", and the slave-server auxiliary program "shusetie".

The one other piece of machinery which had to be fitted in, somehow, was a user interface for talking to *shused*. The actual user interface was a somewhat secondary concern, especially since it seemed that there might have to be more than one, but we needed a way to talk to the daemon. To simplify implementation and separate the major concerns somewhat, we decided to have a separate "gatekeeper" program, *shusedgate*, invoked by *inetd* as required. The gatekeepers implement whatever authentication of credentials is appropriate, and then pass commands to the daemon and responses back, communicating with the daemon via a set of FIFOs. Apart from network interface and authentication, their role is to enforce timeouts on interaction and shield the daemon from possible interference by uncooperative users.

When network connections get involved, security is an obvious worry. The right way to deal with this in the long run, clearly, is with encryption. As a stopgap measure, since Sheridan already relied heavily on NFS being trustworthy[2], we decided that network

[2]Not necessarily a safe assumption, but that's the way it was.

```
name      spencerh
passwd    76hgfu645fmvt
passwd@   806860944 (Thu Jul 27 12:02:24 1995) spencerh
uid       8172
gid       15
home      /home/apollo/it/spencerh
shell     /bin/sh
server    it
schema    n
status    active
status@   807477421 (Thu Aug  3 15:17:01 1995) root
fullname  Henry Spencer
workphone 1-416-555-4444
office    E108
mailname  henry.spencer
changed   807988782 (Wed Aug  9 13:19:42 1995) root
```

Figure 1: User database entry

connections would pass only ‘‘hey, wake up and look at this’’ requests, with all crucial information being passed via the file system.

Although the initial user community was to be mostly the system administrators, a fairly flexible permission scheme was clearly desirable. At one extreme, users had to be able to change their own passwords (we briefly considered using more traditional mechanisms for that, but decided that having Shuse handle *everything* was simpler than dividing the responsibility). At the other extreme, the system administrators had to be able to make fairly arbitrary changes. And there are a variety of interesting levels in between, such as help-desk personnel, who should be able to interrogate the database and do some limited operations like changing passwords but should not be permitted to do more drastic alterations.

To provide a flexible permission scheme, we tag each daemon operation with a ‘‘category’’, and a control file specifies which categories of operations are open to which users. At one extreme, a few read-only operations are in category ‘‘harmless’’ and are available to everyone. At the other, arbitrary editing operations and the ability to shut down the daemon are in category ‘‘overlord’’, which is restricted to a small set of users calling only from the central server machine itself.

Implementation Approach

With the design outlined, implementation started. The main constraint on it was that Shuse simply *had* to be functioning for the September 1995 student intake.

With some trepidation, we decided that Shuse would be written essentially entirely in Tcl [4,5]. Experiments suggested that performance would be adequate, and the use of a very-high-level language looked like it would speed up development considerably. Crucial portions could always be re-coded in C if necessary. This basic approach was quite successful in earlier projects [6], and Tcl seemed a better choice than the Unix shell (which tends to be slow and clumsy unless the task at hand is suited to the Unix utilities) or Perl (which tends to be ugly and unmaintainable).

We quickly settled on Expect [7,8], one of the most popular Tcl extensions, rather than ‘‘raw’’ Tcl. Initially, this was done because we anticipated uses for Expect’s ability to start and control other processes. It turned out that Expect also has a number of small amenities which make it a more complete programming environment than Tcl, which was envisioned as a minimal extension language rather than an independent programming language. (For example, Expect can catch Unix signals.)

We used Expect’s special I/O primitives in fairly minor ways (although see the discussion of *shuselace* later). The one area where we used Expect’s facilities more seriously was in calls to slave servers, to invoke *shusetie*. The actual invocation of *shusetie* was done by *inetd* on the slave server, but the daemon did have to be able to make a call across the network. Rather than add primitives for this, or adopt one of the existing Tcl networking extensions[3], we used Expect’s primitives to invoke *telnet*, specifying a non-standard port number to reach *shusetie* instead of *telnetd*. This may sound a little ugly, but in fact it is quite simple and practical, and has the bonus that portability is almost automatic: all the system-specific complications of networking are invisible.

To make Shuse easier to maintain, each of its programs reads in a configuration file as part of startup. Rather than parsing the configuration file and interpreting its contents, the program simply ‘‘sources’’ it, running it as part of the program’s own Tcl source. This makes it possible for the configuration file to contain not only the obvious variable settings – permissions, pathnames, etc. – but also Tcl procedures. As a case in point, the procedure that builds a *passwd* line from a Shuse database entry resides in *shused*’s configuration file, so that arbitrary changes in *passwd* format can be accommodated without diving into the main sources.

Gritty Details

Certain aspects of the implementation posed unexpected difficulties. We anticipated some of these, while others came as unpleasant surprises. (In some cases, we accepted marginally-satisfactory early solutions simply to get Shuse functioning in time; not all of the issues mentioned here were fully resolved for September.)

We had originally envisioned that extracts from the database, such as the *passwd* file or the user lists for *shusetie*, would simply be assembled by *shused* as necessary. It turns out that digging through the whole database every time such a thing is needed is relatively costly. This means that (for example) building a *passwd* file is expensive, and looking up a user by student number or mailbox name is very slow. Moreover, the information rarely changes much, so rediscovering it each time is wasteful.

Shused now builds internal auxiliary databases at startup, and updates them accordingly when relevant information changes. In some cases, the more costly updates are postponed until *shused* appears to be idle. For example, after startup *shused* uses idle time to build a copy of each user’s *passwd* line, and keeps those copies around. This permits pumping out a complete copy of the *passwd* file in a few seconds, when needed.

The single-threaded design of the daemon is awkward when long-running chores have to be done,

[3] Tcl itself acquired networking primitives in release 7.5, early in 1996, but that was about a year late for Shuse.

because an interactive request should not be delayed arbitrarily waiting for such a chore to finish. Long-running chores must either be broken up into small pieces, so that interactive requests need not wait too long, or be farmed out to auxiliary processes to take them out of the critical path entirely.

A particular problem area is that an update of a slave server can be very slow. The bigger slave servers contain thousands of home directories, and merely enumerating them all for comparison with a user list is a slow operation when user load is heavy. The biggest performance problem in early Shuse operations was long delays in interactive requests when *shused* was waiting for responses from *shusetie* running on a slow slave. When the systems were busy, the extremes of the response time were utterly unacceptable.

We briefly considered multi-threading *shused*, but apart from certain practical problems – it's not something Tcl does well – it seemed unnecessarily general for what was, after all, a somewhat specialized problem. We tackled this one from the other end: during startup, *shused* spins off a "flunky" process, dubbed *shuselace*[4], which does all calls to the slave servers. The flunky makes calls to *shused* using (almost) the standard user command interface, with minor special privileges. (Expect's I/O primitives make it trivial for *shused* to listen for input from two sources instead of one.) *Shused* itself maintains a queue of work to be done by the flunky, and provides "user" interfaces which do things like removing one item from the queue. The flunky uses a *shused* command to pick up a work item to be done (e.g., "update server *nova*"), goes away and does it (taking as long as necessary), and then uses another *shused* command to report success or failure. The first version of the flunky was mostly code transplanted intact from the innards of *shused*, and setting it up took only a day or two's work. It was entirely successful, and response time has never again been a significant problem.

During development of Shuse, we were generally preoccupied with the daemon and its auxiliaries, and did not give much attention to the user interface. We obviously needed some sort of command interface to test the daemon, so a simple program that sends the daemon a single command gradually appeared, more as a debugging tool than a finished user interface. Naturally enough, it was fairly promptly pressed into service as a user interface. While it is somewhat inefficient – for bulk operations, one would prefer to be able to send the daemon more than one command at a time – it works sufficiently well that there has been little incentive to replace it. In particular, it is exactly what is wanted for writing scripts.

[4]In retrospect, we should probably have named the slave-server program *shuselace* and the central-server flunky *shusetie*, since the flunky manipulates the slave-server programs rather than vice versa, but it's too late now.

The one additional user interface that had to be provided was a naive-user password changer. We re-implemented the *passwd* command (and *yppasswd* as well) as an Expect script that requests old and new passwords, does appropriate checks[5], and then calls *shused* to make the change. (Naturally, *shused* itself also does some checks before permitting the change!) This required adding another auxiliary C program, 30 lines of code which invokes the password-encryption routines and outputs the result.

The idea of making slave-server updates idempotent, by having *shusetie* compare existing users against a list of users who should be there, was a good one. It turned out to be a bit harder to implement than we expected. For one thing, it's purely and simply difficult to enumerate all the home directories on a server unless the server's directory structures are laid out to make this easy. For another, the comparison approach handles additions and deletions relatively easily, but can't be gracefully extended to handle moving or renaming users. We ended up doing substantial revisions to the structure of both *shuselace* and *shusetie* to implement a more general command facility within them, so *shused* could order specific operations done. Moreover, this involves some relatively fancy footwork to ensure that such operations are not lost if one of the servers crashes at an inopportune moment, and also some slightly more sophisticated authentication to assure *shusetie* that the thing sending commands is really *shuselace*.

One particular problem in the implementation of *shusetie* was disk quotas. The so-called user interface of the quota system is a disgrace to Unix: inflexible, interactive only, and completely lacking in reasonable primitives for system administration. To cap it off, DEC reinvented the wheel here: when they implemented a new filesystem type, instead of extending the existing quota commands to handle it, they added a new set in parallel – with the same crippling deficiencies in functionality, and some unhelpful changes in data format – so that on a mixed system, you may need to edit a user's disk quotas twice, with two different commands, to get them all!

Fortunately, Expect came to the rescue here. Since the quota facility does at least let you chose which editor you want to use to edit the quota data, we originally thought we'd just have it invoke *ed*, which we could drive with an Expect script. In the end, it turned out to be simpler to move some of the intelligence into the editor, so *shusetie* now manipulates the environment to make the quota commands invoke a little customized editor written in Expect. There is still the annoyance of having to do this twice, via two different sets of quota commands, but some careful

[5]We note that it is *vastly* easier to change or improve the is-this-a-good-password test when the program is written in a very-high-level interpretive language!

design of the editing primitives made it possible to do this fairly painlessly.

Successes

The bottom line is: it works. We're still discovering things that need improvement, but the September 1995 crisis was averted, and as of May 1996 the system was managing over 20,000 user accounts. (The predictions turned out to be low – instead of doubling, the user base more than tripled.) Response time is good since the implementation of *shuselace*, and the staff workload for routine administrative chores is declining.

The central-daemon approach is a weak point in theory but it seems to be adequate in practice. Our opinion is that unless unusual requirements are present, it's better to put effort into making a central server reliable than into making the software do without one. Compared to a more distributed approach, a single central daemon enormously simplifies debugging, synchronization, and management.

Using Expect was a big win. We couldn't possibly have met the schedule using C or the equivalent; in fact, we barely met it using Expect. Very few of the problems were a consequence of the interpretive language, and many of the rapid and simple solutions were a consequence of it.

Although Tcl, and hence Expect, is extensible, we did not find it necessary to do this. The option to add language extensions written in C always existed, but in practice we found that the few missing primitives were more easily implemented as separate programs, invoked as needed. For example, the gatekeeper invokes a 30-line C program, which does a *getpeername()* and a *gethostbyaddr()* and prints the result: the name of the host an incoming call is from.

The in-memory-database approach works well. We did end up adding some more RAM to the central server. (We note with some annoyance that conventional system interfaces are too ready to page out seemingly-idle process memory, and don't provide a way to say "let this process make as much of a pig of itself as it wants, and don't page it out unless you really must".) The response time for simple database queries is entirely dominated by the communications arrangements.

The update performance of the file-per-user on-disk database has been excellent, and although it takes several minutes for the daemon to start up and read in all those files, this is a minor nuisance rather than a serious problem. We have occasionally contemplated implementing facilities to dump out the daemon's in-memory database in some form that would permit rapid reloading, given that most daemon restarts are planned, but to date it hasn't been worth the trouble.

The extensible text-based format of the database entries themselves has permitted a number of unplanned additions and amendments. There will surely be more.

While interest in sophisticated user interfaces, e.g., for the help desk, remains, the simple send-one-command interface has been amazingly successful. In particular, an extensive body of scripts has grown up to reflect local policy and frequently-run database operations. We very strongly believe that we made the right decision: do the command-line interface first, leave the fancy graphics for later.

Problems

Not everything went smoothly. Apart from the implementation difficulties mentioned earlier, some broader issues deserve mention.

As one might predict, the customer wishlist changed and grew once an initial system was operating. Things that weren't even mentioned in the original specifications turned out to be major issues that needed substantial reworking of the software. For example, the original design included a very simple facility for automatically executing commands at specific times, vaguely modelled on the Unix *at* command, and this saw such extensive use that some major re-engineering work was needed to make it more practical and efficient.

The original design had little ad-hoc protocols for each communications path. Only the protocol used between the gatekeeper and the daemon was fully fleshed out and pinned down. Since then, many of the paths which originally needed very little sophistication have grown to need the full nine yards; for example, *shusetie* now provides a full command interface to *shuselace*. One thrust of recent work has been to encapsulate the gatekeeper-daemon protocol in a library module, and convert everything to use it; this is almost complete, and has been a definite success.

Telnet connections, while adequate for commands, are suboptimal for bulk data transfer. Early versions of *shused* operations which returned very large amounts of data had mysterious problems with little bits of data loss. Debugging this was difficult, but we eventually established that the problem was in *telnet*, not in Shuse – it would seem that we were overstressing something in DEC's *telnet* or *pty* implementation. As a workaround, the few operations which routinely need to transfer large amounts of data were revised to do the transfer via the file system. The exact cause of the problem was never fully determined, and in fact we suspect that a system upgrade somewhere along the way may have fixed it. The new protocol library checks the length of data transferred in all operations, as a precaution.

We're also interested in the possibility of reimplementing some of the Shuse *telnet* communications paths using Tcl's new networking primitives. While this is of no real importance for Shuse's internal communications, improving the user-interface response time would be nice, and it looks like most of the time spent there is in setup and teardown overhead rather than actual communication.

One area that has not yet been fully sorted out is logging and trouble reporting. After some unsatisfactory early experiences with coordinating multiple log files, the protocol library and some other facilities were extended slightly to let everything send log entries to the main daemon. This has helped, but we still need to do some more work in the area; it's particularly difficult to get satisfactory reporting in cases where final execution of an operation has to be delayed, e.g., because a server is down. Queueing up the work until it can be done is only half the job. It would be useful to have *shused* (or supporting software) maintain a current-status report on each slave server, to make ongoing problems more visible.

As mentioned earlier, response-time constraints and the single-threaded nature of the daemon require that time-consuming internal operations be broken up into smaller pieces. This has gotten easier as experience has accumulated, but that experience really needs to be distilled into a set of library routines that would make it relatively painless. There are a few infrequent operations which would benefit from being split up, but which are still in one piece because it's too much trouble.

The current implementation of Shuse is very much organized around running a single database: the users. In practice, this has been adequate for Sheridan's needs, and extensions in this area have been low priority. For example, Sheridan makes relatively limited use of Unix groups, so the group database is still managed manually. This would be less satisfactory for an installation which made more sophisticated use of group memberships.

In retrospect, the exact split of responsibilities between the software contractor and the Sheridan staff was not quite right. In particular, *shusetie* invokes external shell scripts to do things like creating or deleting user home directories. This works, but it would work better if the scripts were better integrated; in particular, error diagnosis would be improved. The lack of integration is an artifact of the responsibility split, and was largely forced on us by time constraints, but it's still a blemish.

Conclusion

Despite being written in an interpretive language and having to fire up other programs for, e.g., network communication, Shuse works fine and does the job. Performance has not been a problem since minor design errors were corrected, new functionality is easily added, and the system is coping well with databases of 20,000+ users.

Sheridan is happy, and commercial marketing of the software is being explored.

Acknowledgements

Although I wrote almost all of the Shuse software, a number of other people were involved in various ways. Cheri Weaver, then head of Sheridan's system-administration group, got me into all this :-) in the first place. John Barber, her successor, has happily funded ongoing work and enhancements. Simon Galton handled the Sheridan side of Shuse, during its initial transition into production use, with skill and no small amount of bravery ("you're going on vacation *WHEN*, Henry?!?"). Seela Balkissoon, Rob Naccarato, Trevor Stott, and the other members of Sheridan's CSG have patiently used, commented on, and complained about Shuse while it was struggling towards operational maturity.

Availability

The Shuse software belongs to Sheridan College. Times are hard for educational institutions in Ontario, and there is local commercial interest in Shuse, so at this time it is not available free.

Author Information

Henry Spencer is a freelance software engineer and author. His degrees are from University of Saskatchewan and University of Toronto. He is the author of several freely-redistributable software packages, notably the original public-domain *getopt*, the redistributable regular-expression library, and the *awf* text formatter, and is co-author of C News. He is currently immersed in the complexities of implementing POSIX regular expressions. He can be reached as henry@zoo.toronto.edu.

References

[1] Mark A. Rosenstein, Daniel E. Geer, & Peter J. Levine, *The Athena Service Management System*, in Proceedings of the Usenix Technical Conference, Winter 1988 (Dallas), Usenix Association 1988.

[2] Dr. Magnus Harlander, *Central System Administration in a Heterogeneous Unix Environment: GeNUAdmin*, in Proceedings of the Eighth Systems Administration Conference (LISA 94, San Diego), Usenix Association 1994.

[3] Paul Riddle, Paul Danckaert, & Matt Metaferia, *AGUS: An Automatic Multi-Platform Account Generation System*, in Proceedings of the Ninth Systems Administration Conference (LISA 95, Monterey), Usenix Association 1995.

[4] John K. Ousterhout, *Tcl: An Embeddable Command Language*, in Proceedings of the Usenix Technical Conference, Winter 1990 (Washington), Usenix Association 1990.

[5] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley 1994.

[6] Geoff Collyer & Henry Spencer, *News Need Not Be Slow*, in Proceedings of the Usenix Technical Conference, Winter 1987 (Washington), Usenix Association 1987.

[7] Don Libes, *Expect: Curing Those Uncontrollable Fits of Interaction*, in Proceedings of the Usenix Technical Conference, Summer 1990 (Anaheim), Usenix Association 1990.

[8] Don Libes, *Exploring Expect*, O'Reilly & Associates 1995.