Relieving the Burden of System Administration through Support Automation

Allan Miller & Alex Donnini – HandsFree Networks

ABSTRACT

The number of computer users with little or no training continues to rapidly increase. Networks are at the heart of companies large and small. Applications, ever more complex, span intranets and extranets. This all adds up to an increasing burden on system administrators and end user support organizations at a time when there is constant downward pressure on support budgets and a shortage of qualified staff. The result is a technical support crisis with dissatisfied end users, burned out system administrators, and unhappy support teams. The need for support automation is critical as it enables the scaling of effective end user support while minimizing the need for additional resources.

There are significant challenges to automating support, since it is essentially a large collection of special cases. Different methods have been used to achieve this automation, with varying degrees of success. This paper describes a software system that automates the solution of many recurring end user problems, greatly relieving the burden on system administration staff for mundane issues. We describe the architecture of the system, give examples of its use, demonstrate its extensibility, and report on our experience using it in the field.

Introduction

The widespread use of computers in mission-critical functions continues to grow, resulting in their use by ever-larger numbers of technically unsophisticated end users. At the same time, the software installed on them gets more complex as competitive pressures force vendors to incorporate additional advanced features. To make matters worse, the infrastructure itself is becoming inherently more sophisticated: a seemingly simple application such as email involves the use of network connectivity and routing that is far beyond the capabilities of an end user to diagnose and repair. Yet budgets typically do not leave room for additional support needs, as demonstrated in [Tol92]. The increasing demand for support services by end users puts system administrators in a position where they have to solve more and more repetitive issues, in addition to dealing with the more complex enterprise-wide issues they face every day. Often, they are simply not in a position where they can handle this additional burden.

Outside the enterprise, the situation is even worse. With less access to system administration resources, smaller businesses rely on a combination of internal resources and third party service providers. The labor intensive support process used by third party service providers simply does not scale to the level being demanded by current and future use. As evidenced in [DeK99], smaller businesses that cannot afford a full-time system administrator are increasingly dissatisfied with their support. As software becomes embedded in more and more consumer products (such as cell phones or even refrigerators), the problem continues to worsen and is beginning to affect home users as well as businesses.

As a result, there is an urgent need for an increased level of automation in the process of everyday administration, maintenance, and support tasks for end users. System administrators must be freed from the overwhelming load of essentially trivial problems that recur on a frequent enough basis to interfere with the solution of more important and challenging problems. This paper describes a software system that automates the solution of many recurring end user problems, greatly relieving the burden on system administration staff for mundane issues.

Existing Support Automation

One of the indications of the need for automation of end user support is the existence of a range of products in this space. These products all address the need to some point, but have various shortcomings that are addressed by the system described in this paper.

There are a number of products that detect system level events, such as processor faults, and attempt to recover from their effects. These products use a general mechanism and apply it to all instances of the problems being detected. However, the general mechanism really only addresses the immediate cause of the problem event, and does nothing to permanently address the root cause of the underlying problem.

Some products provide automated software updates to customers. This solves problems caused by bugs that are fixed in a later release of the software.

However, many end user issues are not caused by bugs, but are instead usability issues where the software is working as designed but the designers have not foreseen its obscure behavior in a common circumstance. In addition, most problems can be addressed with workarounds that end users can apply immediately, eliminating the need to wait for the next turn of the software development cycle of the product.

Virus scanners are important tools for every system administrator. They detect system level events and changes to files, use a database to drive the detection and removal of viruses, provide reporting functions on virus detection and elimination, and periodically update the database automatically. As we will see, the system described in this paper is similar in some respects to a virus scanner. However, virus scanners have a very specific problem domain on which they focus. They provide no help to a system administrator for the myriad of other end user issues that arise on a daily basis.

"Self-healing" systems provide a facility, either manually guided or semi-automated, to restore some or all files back to a previous state. This can be a powerful device for quickly getting a system back to a working condition. However, it is most helpful in a disaster recovery situation, and is not of much help when a problem must be solved rather than removed, or when a problem has existed for a long time without manifesting itself.

Some "web-enabled support" sites provide a database of known problems and solutions (usually called a "knowledge base"). [Mic00] is a good example of a well-built knowledge base. While knowledge bases have demonstrated their value in providing quick access to vital information for system administrators, the concept of their direct use by end users leaves much to be desired. Most end users have difficulty finding and understanding information in a knowledge base, and may actually end up doing more harm than good by attempting to apply inappropriate solutions. In addition, moving the task of system management into the hands of an inexperienced end user is not a very popular option in an enterprise environment where "time is money."

Help desk software is a key component of an enterprise solution for end user support. Effective help desk software can increase the efficiency of an internal support group and therefore reduce some of the burden on system administrators. In addition, some help desk software can assist an internal support group in building a database of problems and solutions, which can be extremely useful to administrators. However, the software is of little direct use to those system administrators by itself.

Some help desk add-ons allow end users to contact a support group electronically, using email or chat, and facilitate problem resolution through the use of advanced diagnostics. Some of these products also feature limited automation of problem resolution activities and remote support capability, allowing a support representative to take control of an end user's desktop to resolve a problem. This removes some of the potential for human error in diagnosing and resolving the problem. These tools can certainly help a system administrator maintain a larger number of end users more efficiently. However, even with these tools, each end user issue requires individual attention, so

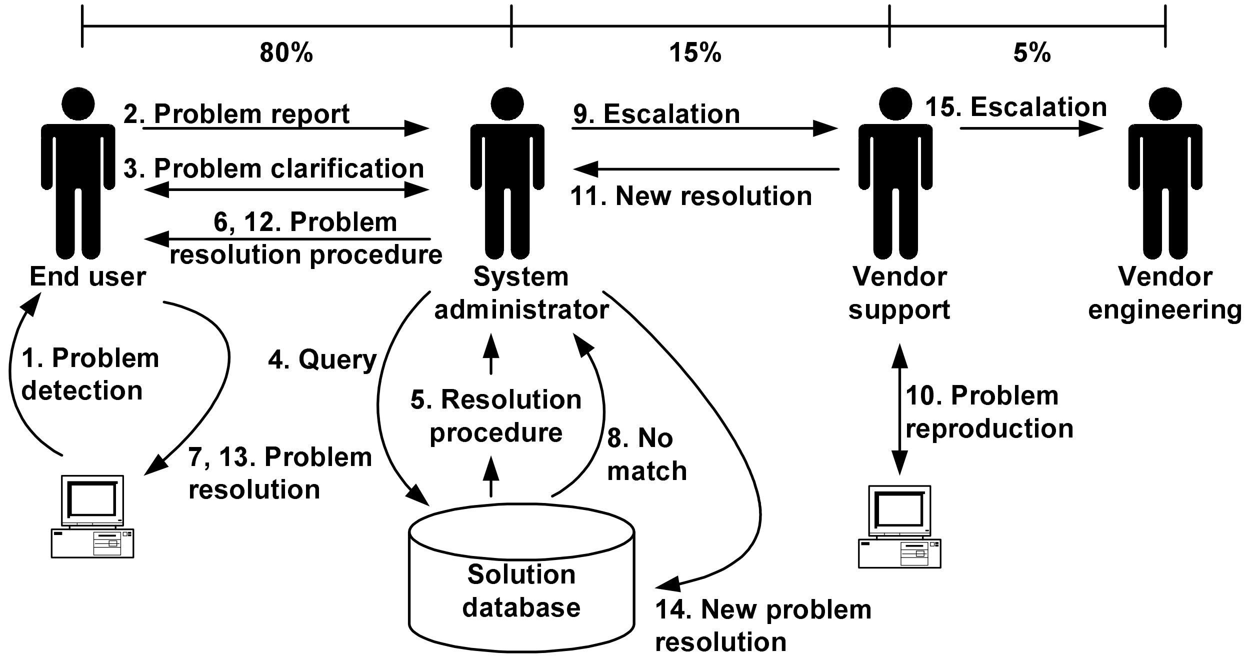

Figure 1: Support process.

the system administrator is still forced into the inefficient role of providing repetitive end user support on common problems.

"Expert marketplaces" match up end users with consultants who are bidding services to solve desktop computer problems. This is an interesting "free market" approach to providing end user support by outsourcing it to a widely distributed support staff. However, it does suffer from somewhat inconsistent results, due to the variety of talent bidding the support services. It is easy to imagine a situation where multiple consultants would inadvertently provide conflicting point solutions to a system wide problem, merely interfering with the efforts of the system administrator who is ultimately responsible for the overall solution.

Finally, a number of "support portals" are available that feature online access to some combination of the services listed above. These provide convenient "one stop shopping" for those services, but do not otherwise enhance the automation or scalability of the services themselves, and are typically geared to a lower level of expertise than that of a system administrator.

In summary, while these approaches are all valid and help with some aspect of the end user support issues faced by system administrators, none of them really shorten the administrator's lengthy task list by automatically detecting and resolving simple recurring problems without any manual intervention. The system described in this paper addresses many of these shortcomings and provides a true solution for saving time and lightening the load on the system administrator.

Automating Support

Figure 1 shows the traditional process for handling end user problems, as exemplified in [RSA00] and [Smi97]. The process starts when an end user determines that a problem is happening. The user contacts[1] the system administrator. Through a discussion with the user, the administrator clarifies the nature of the problem, then searches through a database of pre-defined problem descriptions for a match. In many cases, the "database" is simply a mental list of common problems and solutions. It can also be an informal written list, or if the system administrator is using help desk software to manage the process, it may be an actual database. If a match is found, the administrator communicates the resolution to the user, and assists the user in applying it. [Etc98] describes that this process works well, solving as many as 80% of the problems encountered by end users [Sla00, Gil00, Sup00, Sch00, Hon00, CWS99, Loc00]. For the remaining 20% or so, the system administrator must become directly involved in the diagnosis and solution of the problem, and may need to escalate the process to include the software vendor. If the problem is escalated, the vendor uses product-specific knowledge to reproduce the problem and attempt to resolve it. If the resolution is successful, the procedure for resolving the problem is communicated back to the system administrator, who in turn communicates the resolution to the end user and assists in its application. In addition, the system administrator now adds this new problem and resolution to whatever "database" is used for solving common problems. In that way, new instances of the problem can be resolved more quickly, without needing either direct involvement or the assistance of the vendor. The 5% or so of problems that cannot be resolved by the vendor's customer support are escalated internally to the engineering team at the vendor, where development resources are brought to bear on diagnosing and resolving the problem. In this way, the efforts of the system administrator and the vendor's customer support group play two critical roles in the process: keeping the "database" populated with relevant problems and solutions for common recurring problems, and providing qualified, valid bug reports to the vendor's engineering staff.

Clearly, the amount of human effort involved in Figure 1 makes it a very labor-intensive and error-prone process. To make things worse, the "database" in Figure 1 may be simply mental notes, so the system administrator may be the only person who can drive problem resolution. Even with a more formalized process, the quality of the response is very dependent on the ability of the person using the database, whatever its implementation.

Figure 2 shows the process for automated support. A software client manages the process for the end user. When the client detects that a problem is happening, as described later, it searches a local database for a match to the symptoms of the problem. If a match is found, the database contains executable code that the client runs in order to resolve the problem. Since the database is functionally equivalent to the database in Figure 1, this solves as many as 80% of the problems encountered by the user. For the remaining 20% or so, the client forwards a detailed description of the circumstances surrounding the problem to the system administrator, who is made aware of the problem early on. At that point, the administrator may choose to solve the problem or may instead want to escalate it to the vendor. In either case, the information from the client can be used to diagnose, reproduce, and solve the problem. When a solution is found, either the system administrator or the vendor codes and tests a solution for the problem and adds it to the master database. This master database is used to update the local database, which the client then uses to resolve the problem. The 5% or so of problems that cannot be resolved by the system administrator or vendor's customer support group are escalated to the

[1]This could be a personal contact or a telephone call. Increasingly, other forms of interactive electronic communications are being used.

vendor's engineering staff. The system administrator and vendor's support group continue to play the same two critical roles as in Figure 1: keeping the master database populated with relevant problems and solutions for common recurring problems, and providing qualified, valid bug reports to the vendor's engineering staff.

It is worth pointing out that some problems cannot be detected, and some solutions cannot be implemented, by the system outlined in Figure 2. The client cannot automatically detect problems that completely disable the system, such as a faulty power supply, since the system cannot run the client. The client cannot automatically implement a solution that requires physical modification of the system, such as replacing memory. Likewise, the client cannot automatically detect problems that are not amenable to software detection, such as poor quality in printer output. Finally, the client cannot automatically resolve problems that are caused by a misunderstanding on an end user's part, such as inability to set the margins in a word processor. However, two powerful techniques can be used to overcome some of these limitations. The first is the use of communications between clients on multiple machines. Even if a problem cannot be detected directly on the machine where it occurs, its external effects may be readily identified on other machines in the network. The second is the use of an interactive dialog with the user of the machine. Even when the client cannot test or modify certain aspects of its environment, it may be able to instruct the user to do so on its behalf. In the end, even if the client cannot completely diagnose or resolve a problem, the information it gathers in doing so will be immensely helpful to the system administrator ultimately responsible for the problem resolution. Our analysis, described later, of the most frequently occurring problems in the Microsoft Knowledge Base, [Mic00], indicate that software configuration issues cause the vast majority of these problems. This means that the kinds of problems the client cannot address make up a relatively unimportant fraction of those that actually happen. Anecdotal evidence also seems to support this conclusion.

Two important features of automated support are evident in Figure 2. The first is that up to 80% of the problems that occur on the end user system are resolved quickly and accurately, without any human intervention. The user may not even be aware that the problem existed in the first place. The second is that the tasks of the system administrator are driven by the appearance of new problems, not by the appearance of new users. This means that the part of Figure 2 that needs to scale up the most to handle more end users is the update of the local database from the master database. An increase in the number of users does not require a proportional increase in the system administration group, so large numbers of users can be supported without a huge staff.

There are significant challenges to automating support. By definition, the task is a large collection of special cases, but software tends to work best with small sets of algorithms applied to large numbers of uniform tasks. In addition, the applications that are being supported cannot implement the support themselves, since they are presumably malfunctioning.

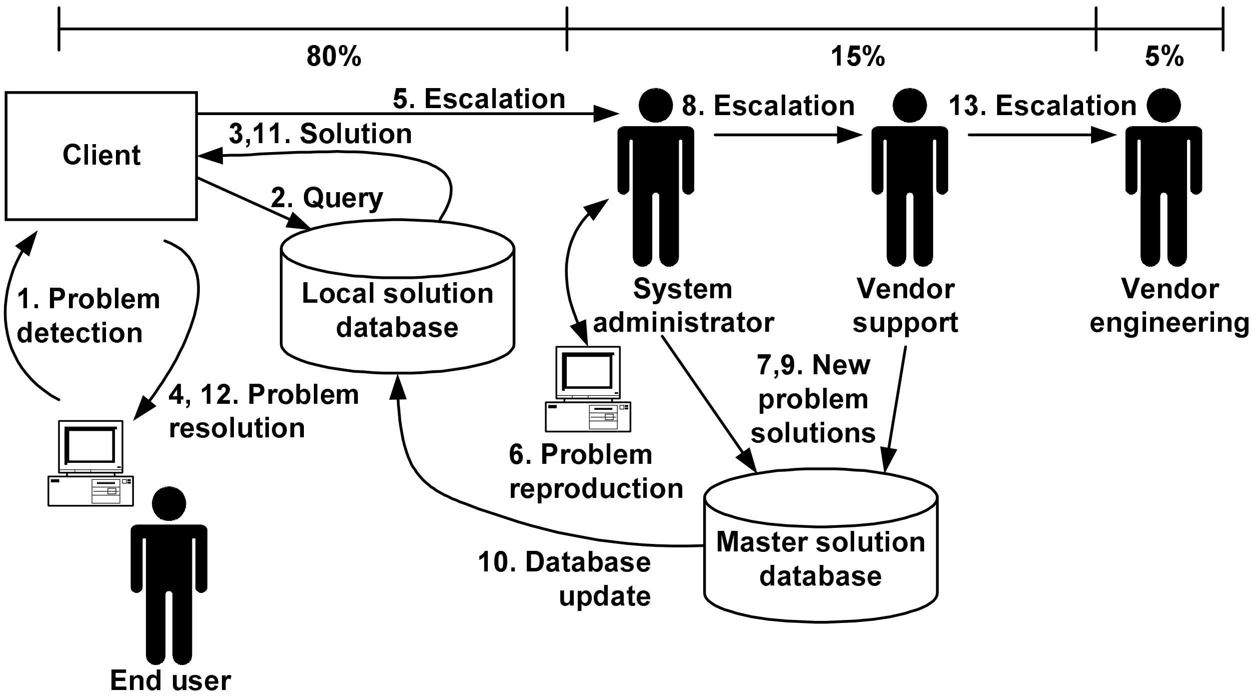

Figure 2: Automated support.

Since nearly any aspect of system operation can be part of the problem, and part of the solution, the automated support tool must be as independent as possible from the system and its applications. At the same time, it must be aware of most of the details of the system operation.

These factors make the job of automating support difficult, but not impossible. Database techniques can be used to manage the large number of special cases. Both the diagnosis of problems (“what happens when you run...?”) and their resolution (“now change the setting...”) are essentially software tasks that can be automated. Combining these two considerations, the implementation of automated support works well with a database containing executable code for the diagnosis and resolution of problems. The size of the database is strongly affected by the “80-20 rule”: 80% of end user issues that arise can be solved using only 20% of the entire universe of solutions. (Note that this is a little different from the 80% and 20% mentioned above, which referred to the fact that 80% of user issues can be resolved using a database.) The 80-20 rule is well known in support circles [Che99, Che99a, Gia99, Lan00, Ste99, Sum98]. The implication is that a relatively small database can be extremely effective in resolving user issues. We now describe the use of such a database in an automated support system provided by HandsFree Networks.

System Architecture

The heart of the system is the client that runs on each end user machine in the facility. Figure 3 shows an overview of the client. The database contains a collection of scrips,[2] each of which contains executable code to diagnose and resolve a single problem. A scrip consists of four major parts: initialization, configuration, symptom, and solution. These major parts are described in more detail later in this paper.

[2]The term “scrip” is borrowed from the medical profession, where it is the working jargon for a prescription. Like these database entries, pharmaceuticals are used both for diagnosing and resolving problems. There is also a connection to the word “script.”

Event detection drives the application of scrips to problems based on system level events such as window creation, user input, processor faults, process creation and termination, device notifications, and timer expiration. The event detection module efficiently determines which, if any, scrips should be run as a result of a sequence of system level events. It is designed to very quickly dismiss events that do not trigger a scrip. Since the resulting conceptual model of a scrip is an interrupt service routine rather than a polled service, a scrip only uses system resources when a problem is detected, thereby minimizing its impact on the system.

The execution scheduler is responsible for dynamically loading and executing the code for a scrip on an as-needed basis. It is in charge of managing the synchronization of scrips that are running in multiple threads, as well as passing information from detected events to the scrips. The database also contains procedures that are dynamically loaded when they are called from a scrip; the execution scheduler manages the loading and execution of these procedures, as well as the passing of parameters into and out of the procedures.

Primitives are a library of pre-defined routines that provide a convenient (and, where possible, system independent) interface to operating system functions. They also provide an interface to common utilities such as memory allocation and string manipulation. Scrips and procedures in the database use primitives to accomplish most of their work. From the perspective of a scrip, the interface to a procedure is the same as the interface to a primitive, so procedures in the database serve as a convenient and flexible way to extend the primitives.

Scrips use the persistent state mechanism to keep information that must persist across multiple invocations of the scrip and machine restarts, as well as information that must be accessed by more than one scrip. Variables in persistent state have a namespace that limits their scope, so a scrip author can freely

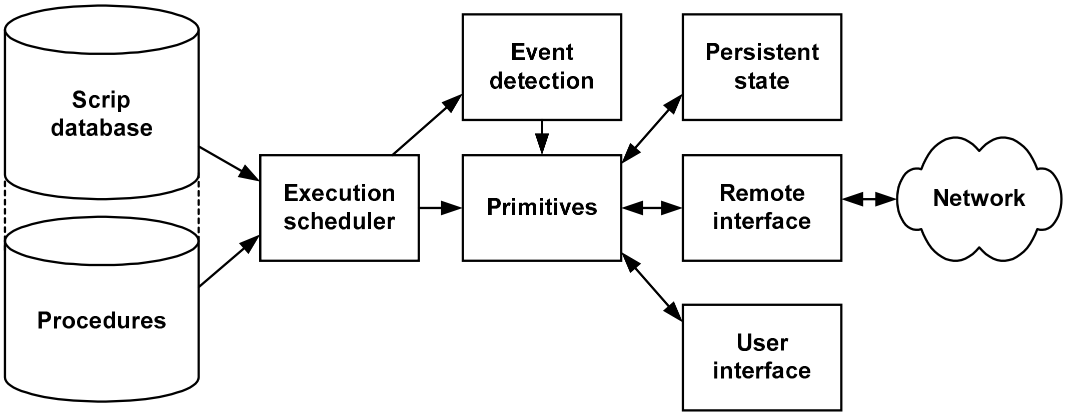

Figure 3: Client Architecture.

define and use a persistent state variable without concern for interfering with the variables in other scrips.

The remote interface allows scrips to run code with equal ease on any machine in the network, including the machine on which the scrip is currently running. This is accomplished by making every call to a primitive specify the machine where it should run. An example of this is described in more detail later in this paper.

A key feature of the client is its portability to multiple environments. It has been designed from its inception to have well-defined system dependent and system independent modules with consistent interfaces. One of the authors has architected three major portable software systems in the past (described in [Vir97, Ter00, CAS87]), and has brought relevant experience to bear on this system as well. The current version of the client runs on many implementations of the Win32 API (Windows 95, Windows 98, Windows NT 4, Windows 2000, and Windows Me), and a Linux version is presently in development.

Much of the value of the system comes from the fact that the scrip database is updated on a regular basis, since end user issues change regularly as new products are introduced and new problems are discovered. As a matter of fact, one of the greatest benefits of implementing the client using a database architecture is the flexibility in providing this update. For example, the scrip database can be implemented as a central database, or a central database with local caching, or a distributed database, or a redundant database, or any combination of those options. In this application, a problem that affects network connectivity may isolate the client from the network, so a standalone local database is desirable. A simplified representation of the default product configuration is shown in Figure 4. A centralized master scrip database contains all available scrips. At the system administrator's facility, a local scrip database stores the subset of the master scrip database that is relevant to the local hardware and software configuration. At a regular interval (about once a week), the client on one of the administration machines initiates an incremental update of the local database to retrieve any new or modified scrips. All the machines at the facility access the local scrip database for the latest versions of all scrips. In addition, they also have a small database of scrips to solve connectivity that is replicated on every machine. If a machine cannot access the local scrip database due to a connectivity issue, this database helps to solve that problem first. Note that both the master scrip database and local scrip database can be replicated for improvements in reliability and performance.

The implementation of security in the system is of paramount importance. Since the primary function of the software is to transmit and run executable code, it would be an ideal virus infection vector. To prevent this sort of abuse, all network access goes through the remote interface and uses the HTTP protocol. The remote interface implements SSL as described in [Fre96]. For maximum security, it can be configured to require both client and server authentication. This encryption protects system integrity by preventing malicious attacks on the code executed by a client, and also protects data security by preventing passive observation of the data communicated between two clients while executing a primitive remotely. In addition, all scrips are digitally signed, and the engine can be configured to only run scrips from a list of trusted providers. Even without strong (or any) encryption, the digital signatures prevent the execution of malicious code by a client, since the attacker can forge a scrip but cannot forge its signature. This protection works for worldwide deployment of the database,

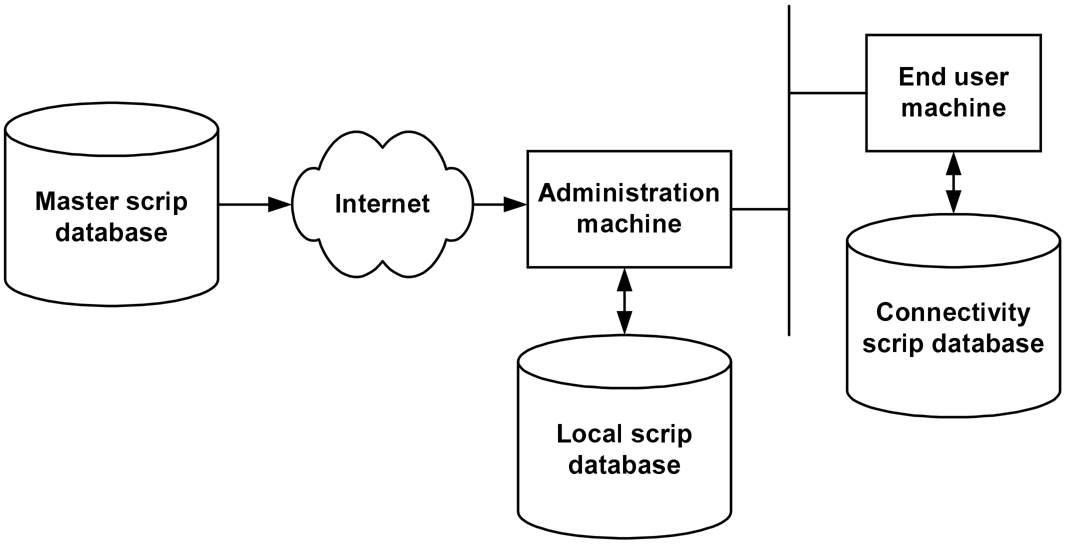

Figure 4: Scrip databases.

since export restrictions on strong security only apply to data encryption, not to digital signatures.

Interestingly enough, the connotation of a ‘‘certificate authority’’ for the SSL security in the client is somewhat different from the normal one. In secure e-commerce transactions, a certificate authority is simply declaring that the identity of the entity being certified is actually correct. A certificate authority for scrips, on the other hand, is declaring that the entity being certified is one that can be trusted to produce reasonably stable and desirable scrips. Presumably, the certificate authority provides this service by verifying that the entity has reasonable QA procedures in place, doing periodic audits of the software production facility, and doing random independent code reviews and testing of the scrips themselves. This is quite a bit more complicated than simply verifying identity, but is worth correspondingly more as a value added service. HandsFree Networks plans to provide this service as part of its product offering, but also expects that third party providers will provide a similar service.

Problem Resolution Process

Referring to Figure 2, the problem resolution process starts when the event detection mechanism recognizes a certain sequence of system level events that corresponds to an entry in the scrip database. It puts together relevant information about the system level events into an ‘‘event’’ that is passed to the execution scheduler, along with an identifier indicating which scrip is being triggered.

The execution scheduler decides when the triggered scrip should be run and loads the scrip symptom from the database. It runs the code for the symptom, which does any kind of verification needed beyond the trigger to ensure that the problem is indeed present. If the symptom indicates that the problem is present, the execution scheduler loads the scrip solution from the database and runs the code for it.

When a problem cannot be resolved locally, an escalation process prevents the situation where an end user is left without a solution. Escalation can be initiated by the failure of a solution, which is detected when a scrip continues to diagnose a problem even after its resolution has been applied. Escalation can also be initiated by a ‘‘diagnostic’’ scrip, which detects a general error condition that does not have a specific solution supplied by another scrip. A set of about twenty diagnostic scrips is used to detect errors that are not resolved by the rest of the scrips. The first step of the escalation process is to initiate an update of the local database from the master database. In the most common case, the problem is a relatively new one that has been discovered and resolved since the last periodic update of the local database. For example, it might be a compatibility problem caused by installing a new version of a popular program. In this case, the new scrip to resolve the problem is retrieved and applied. If, on the other hand, the problem is being encountered for the first time, the second step of the escalation process is to gather relevant information about the state of the system and the error and forward it electronically to the system administrator. This is typically much more complete information than the system administrator usually gets from end users. The administrator can then either reproduce the problem locally to determine its resolution, escalate the problem to the vendor, or simply contact the user directly and resolve the problem through more traditional means. If the user is not local, the system administrator can initiate a ‘‘remote service’’ session, allowing the use of the remote interface module of the client to look at specific system state and apply problem resolutions. After understanding and resolving the problem, either the system administrator or the vendor’s customer support staff initiates a process to add a new scrip to the master database. If the problem cannot be resolved and is escalated to the vendor’s engineering staff, it follows the same support process that it would for non-automated support. Once a resolution is available, however, the system administrator or vendor adds a new scrip with the resolution to the master database. In any case, once the new scrip is in the database, the same problem is always resolved on all end user machines without further manual intervention.

The software also provides a valuable service for administrators who are managing multiple end user sites by providing notification of all detection and resolution activities. This can be used simply for accounting purposes, so that the administrator can easily provide activity reports to management demonstrating the ongoing value of the system administration staff. It can also be used for isolating trends that indicate a particularly troublesome hardware or software component at a particular site, leading to a replacement of that component and a resulting improvement in stability. Finally, it can be used to provide greater visibility to the system administrator. For example, the decision on when to upgrade an application to a new version is typically made on a fairly arbitrary basis today. However, with this notification, the administrator is aware of the number of end user problems occurring that could be solved by such an upgrade. If the number of such problems is small, the upgrade can be postponed, but if it is large, the upgrade might be accelerated. Since most users rarely, if ever, report this kind of information, it is extremely valuable to system administrators. Detailed reporting of problem detection and resolution activities also gives system administrators leverage when dealing with vendors’ customer support groups, since it minimizes the frustrating ‘‘finger-pointing’’ that typically goes on when problems occur in the multi-vendor computing environments that make up nearly all facilities.

This reporting mechanism is very configurable. It can be set up on a scrip-by-scrip basis to report immediately or collect the reports into a single log that is sent on a periodic basis. An email interface provides the simplest, most convenient means of reporting. For system administrators who are maintaining a larger facility, a web-based interface can log reports by adding entries to a database. The administrator can then use a browser interface to peruse the database, spot trends, and manage user issues remotely.

Scrips

Each scrip consists of four parts: initialization, configuration, symptom, and solution. The symptom and solution parts have already been described; both parts are executable code that is loaded dynamically and run by the execution scheduler. The code for the symptom does a conclusive test to see whether the conditions for the problem actually exist. This is required since the events triggering a scrip may not completely define the situation for the problem. For example, the trigger may be an error dialog, but it may be necessary to look at some configuration data (in files or the registry) to determine whether the known problem is actually happening. The code for the solution is the actual implementation of the solution. It can be extremely simple, for example, a solution that solely involves updating some configuration information. It can also be quite complex, for example, a solution that has to try several different resolutions to see if any of them work, and must restart the system (and coordinate this process with the end user) after each try.

Since the execution scheduler needs to set up the interface between scrips and primitives (and also between scrips and procedures) at runtime, the procedure interface across the boundaries of scrips must go through a thin interface layer that ''interprets'' the calls and returns. This has an impact on the performance of scrip execution; however, scrip execution happens at a low frequency and is not time critical, so the overall impact is negligible. On the other hand, one large benefit of this architecture is that symptoms and solutions for scrips can be implemented in any language for which it is possible to implement this thin interface layer. Current implementations of scrips use the ''C'' programming language, but this is mostly a matter of convenience, and interfaces for popular programming languages such as Perl (see [Wal00]) are being developed.

The initialization part of a scrip has several functions. It registers the scrip with the execution scheduler, specifying characteristics about the scrip such as other scrips with which it must be mutually exclusive (cannot run at the same time), and other scrips upon which it depends. Most scrips can work with a default configuration, but more complex ones can have specialized requirements. Initialization of a scrip also defines the persistent state variables used by the scrip. Finally, the initialization defines the conditions under which the scrip is triggered. This is done by specifying the conditions on a sequence of a few fundamental events. For example, a scrip can register itself to be triggered when a certain executable is run with a specific command line, and then a dialog box is created by the resulting process with a given title.

The timing and order of scrip initialization is left unspecified. For a client that needs to implement completely dynamic configuration, the scrip initializations can be run sequentially as the client is starting. For a more conventional client (such as one used in desktop support), the scrip initializations can be run as a preprocessing step whenever the client is installed or the scrip database is updated. The results of the initialization are saved in runtime tables that are read during client startup to provide fast initialization.

The configuration part of a scrip allows the scrip author to create user-defined options that control the function of the scrip. This section specifies classes of input fields, such as radio buttons, check boxes, and text strings. Each input field has a default value and a persistent state variable associated with it. The user interface module presents the configuration information as XML, in a form that can be displayed and modified by a browser. Once this setup is complete, the client uses it to initialize the referenced persistent state variables. When the scrip runs, it uses the values of those variables to control its operation. In this way, the scrip author has great flexibility in allowing customization of the operation of the system. This facility also allows customization of ''overall'' features of the system, since the persistent state variables that a scrip configuration modifies are not limited in scope to variables that affect a single scrip.

There are actually four general types of scrips, although these types are all implemented the same way, so the categorization is a conceptual one rather than an operational one. The scrips that have been emphasized so far are those that solve a specific known problem with a well-defined and tested solution. Another type of scrip is one that detects and resolves problems that arise as a result of end user operations, such as installing a new piece of hardware or software. Still another type of scrip is the ''diagnostic'' scrip previously mentioned that detects a general error condition that does not have a specific solution supplied by another scrip. The fourth type of scrip is the ''preventive'' one that performs routine administration and maintenance functions that can be automated. Clearly, this last type of scrip is immediately useful to a system administrator, even in a system that is functioning perfectly.

To simplify the management of scrips, and to allow them to be conceptually classified, a tree-structured descriptive hierarchy is maintained. This is similar in concept to the hierarchy that is maintained for information accessed by a search engine such as

Yahoo! (see [Yah00]). The hierarchy is expanded as needed when new scrips are developed, and a single scrip can be in more than one place in the hierarchy. The hierarchy is represented in the database containing the scrips, and the user interface provides a facility for navigating the hierarchy to get to the configuration of individual scrips within it. Groups of scrips within a single branch of the hierarchy can be managed as a unit. For example, there are a group of scrips that implement basic functionality within the system, such as sending periodic logs and updating the client software to newer versions. These scrips are required for proper system operation, and are all part of the group of ''system'' scrips within the hierarchy. Implementing much of the system functionality using scrips makes it convenient to update the basic operating functions of the system without requiring software upgrades in the core software, resulting in a more reliable and flexible system.

One of the key design goals in the architecture of the scrip database is to allow authoring of scrips at a multitude of facilities without any explicit coordination. In this way, local expertise with certain types of problems can be captured and disseminated without having a central authority as a bottleneck. The namespace for persistent state, along with the self-organizing features of the scrip initialization and the descriptive hierarchy, combine with the modular nature of the scrips themselves to make the promise of this widely distributed development a reality.

Example

A concrete example of a scrip will help to understand many facets of the system. This example scrip solves a problem that is taken from the direct experience of one of the authors, a problem that recurred on a regular basis and was extremely annoying.

The underlying cause of the problem lies in the shortcomings of the Microsoft Windows print server architecture. It would not arise in a purely Unix environment, but most system administrators recognize the necessity of managing systems with the popular Windows operating system, and certainly concur with the desire to quickly and efficiently handle the many problems it introduces.

This particular problem manifests itself in this way: the facility has a Hewlett Packard LaserJet 4 printer, driven by one of the file server systems running Windows. Whenever a new system is installed, the printer is added as a network printer. Usually, everything works correctly, but occasionally someone prints a document and gets a dialog reporting that the document is ''too complex'' to print. However, if the document is transferred to another machine and printed from there, it prints with no problems.

This problem is acutely frustrating for the end user, because there is no apparent difference in the two systems (same printer driver, same document, same operating system), yet the operation works correctly on one system but not on the other. In addition, the workaround of printing the document from another system is extremely inconvenient, because it disrupts the workflow of two people: the person trying to print the document and the person whose system must be used in order to print it. As a result, the person with the problem document usually ends up trying to ''tweak'' it so that it isn't ''too complex'' to print, resulting in wasted time and an inferior document.

The cause of this problem is the fact that the printer drivers for the HP LaserJet 4 are installed with a default configuration of 2 MB of memory in the printer, even if the printer has more memory. This configuration must be manually changed to reflect the actual amount of memory in the printer, or else the printer driver will not be able to format the document to print because of this perceived lack of memory. In addition, the printer driver has a feature called ''page protection'' that must be turned on in order for the driver to accurately estimate the memory that is needed in order to print a particular document. Unfortunately, these two configuration changes must be made every time the printer driver is installed. Since end users may be installing these drivers themselves, it is difficult to set up a process that insures the application of these configuration changes. The situation is further complicated by the fact that a long period of time may elapse between the installation of the driver and the appearance of the problem. Justifiably, the reaction of the end user is that everything has worked perfectly for a long period of time, so there must not be any problem with the printer setup.

This problem is ideal for solution with a simple scrip. Doing so provides a systematic way to represent ''institutional'' knowledge that is typically maintained informally by the system administration staff. The problem has a relatively simple method for recognizing it, a fairly straightforward means to test for its existence, and a clear procedure for solving it. It is interesting to note that this problem, like many that are addressed by system administrators, is not technically a ''bug'', in the sense that the software is working as designed. It is really more of a usability issue.

The scrip for this example will be presented using ''C'' as a programming language. The details to translate it to any particular programming language are fairly mechanical. The scrip initialization is shown in Listing 1. The initialization starts by registering the scrip with the execution scheduler, giving it a title and the default execution mode, which indicates that only one instance of the scrip should be run at a time. Then it sets up a trigger with the event detection mechanism, indicating that the scrip should be run whenever a dialog is created with the title ''HP LaserJet 4'' and the dialog text ''Document too complex''. Note that the first parameter for each primitive indicates the machine where the primitive should be run. In this case (and in all initialization), the primitives are run on the current machine, which is indicated by the constant CUR.

The symptom for this scrip is shown in Listing 2. The symptom gets the value of a specific registry key that contains the settings for the printer driver. If the registry key does not exist, then the scrip does not apply. It checks to see if the settings are the default ones for the driver, which are represented by a specific 310-byte binary value. If so, then this scrip applies, because it is designed to solve the problem where the default value has not been updated to match the actual properties of the printer. Note that once again, the primitives are all run on the current system. Also note that the parameter to the scrip is a data structure describing information about the circumstances triggering the scrip, but in this case that information is not needed to determine the applicability of the scrip.

The solution for this scrip is shown in Listing 3. The strategy for the solution is to assume that the printer driver settings are correct on the machine where the printer is installed, and copy the settings from that machine. It uses a registry key to get the name of the remote printer, then extracts the name of the machine. It converts the name into the internal representation of a machine identifier, then gets the registry key for the printer settings. This is an example of executing a primitive on a remote machine. The same primitive is used to get the printer settings on the remote machine as was used to get them on the current machine in the symptom, but the first parameter indicates the remote machine rather than the current machine. Finally, the settings found on the remote machine are used to update the settings on the current machine. If at any point, the solution cannot proceed, it returns FALSE, indicating that it has failed. This failure will initiate the escalation process previously described. If the solution succeeds, then a successful problem solution will be logged as described previously.

Database Feasibility

One important question about the system that naturally arises concerns the feasibility of creating and maintaining the database of scrips. We have done significant research to address this question. As mentioned previously, the 80-20 rule indicates that the database size need only be about 20% of the total universe of problems being addressed. Data presented in [All99, Etc98, IHS00, ITS99, and Rea99] lead us to expect that 80% of all end user issues will be resolved

```
extern void Init(void)
{
    /* Register the scrip with the execution scheduler. Use the
       default setup: only one instance of this scrip at a time. */
    RegisterScrip(CUR,
        "Update HP LaserJet 4 setup to correct Document Too Complex",
        STANDALONE);
    /* Set up the trigger for the scrip. */
    SetDialogTrigger(CUR,
        NULL,                      /* owning process */
        "HP LaserJet 4",           /* title */
        "Document too complex");  /* text */
}
```

Listing 1: Code for script initialization.

```
extern BOOLEAN Symptom(TRIGINFO trig)
{
    char *setupVal;
    BOOLEAN exists, compare;

    /* Check to see if the driver is installed, and get setup. */
    GetRegistryBinary(CUR, &setupVal, &exists,
        "HKEY_LOCAL_MACHINE\\System\\CurrentControlSet\\Control\\"
        "Print\\Printers\\HP LaserJet 4\\Default DevMode");
    /* If this driver isn’t installed, the scrip doesn’t apply. */
    if (!exists) return FALSE;
    /* See if it is still the default setup */
    CheckMemoryEqual(CUR, &compare, setupVal, DEFAULTVAL, 310);
    /* If the comparison is equal, the scrip applies. */
    return compare;
}
```

Listing 2: Symptom code.

by this database. We have extensively characterized the Microsoft Knowledge Base [Mic00], focusing on the major products including operating systems, office productivity applications, groupware, and Internet products. Using tools such as FAQs to identify the most frequently encountered problems, we found that a database with 2,500 to 3,000 entries should provide resolution of a large percentage of the recurring problems encountered by end users in a Microsoft Windows environment.

After the database is built, it needs to be maintained on an ongoing basis. Figure 5 shows the number of new entries as a function of time for the category ‘‘Windows 95’’ in [Mic00]. Except for an 8-month window around the operating system release, it shows about 25 new entries every month. Note that this is a pessimistic estimate, since not every entry in [Mic00] represents a problem, and even the ‘‘problem’’ entries do not all represent solutions that require automation. By way of comparison, Figure 6 shows the number of new entries as a function of time for the category ‘‘Windows networks’’ in [Mic00]. This graph has no particular peak around a product release, but shows about 10 new entries every month.

It seems reasonable to say that even an extremely pessimistic assumption for new problem incidence will be less than 80 problems per month. Our initial work has shown that coding and testing a scrip takes less than four work days, so fewer than twenty full time scrip programmers can maintain the database.

In a way, virus scanners serve as an ‘‘existence proof’’ for the feasibility of the scrip database. The effort involved in obtaining, analyzing, and producing a solution for a virus is not too different from the effort involved in understanding a problem and creating a scrip for it. Current virus scanners have databases of about 45,000 viruses, so the effort involved in creating and maintaining the scrip database should be relatively small by comparison.

Field Experience

An initial version of the software described here is currently deployed and operational at a number of customer sites. It has been running at one site, a public relations firm with about 15 nodes, since February 2000. Starting in March 2000, major functionality enhancements were released to that site on a weekly basis, and in April 2000 installations began at additional customer sites. As of the time of this writing (September 2000), the software is running at ten customer sites.

```
extern BOOLEAN Solution(void)
{
    char *port;
    char *machineName;
    char *setupVal;
    MACHINE mID;
    BOOLEAN exists;
    /* Get the machine where the printer is installed. */
    GetRegistryString(CUR, &port, &exists,
        "HKEY_LOCAL_MACHINE\\System\\CurrentControlSet\\Control\\"
        "Print\\Printers\\HP LaserJet 4\\Port");
    if (!exists) return FALSE;
    if ((port[0] != '\\') || (port[1] != '\\')) return FALSE;
    GetSubstring(CUR, &machineName, port, 2, '\\');
    GetMachineID(CUR, mID, machineName);
    /* Get the printer setup from that machine. */
    GetRegistryBinary(mID, &setupVal, &exists,
        "HKEY_LOCAL_MACHINE\\System\\CurrentControlSet\\Control\\"
        "Print\\Printers\\HP LaserJet 4\\Default DevMode");
    if (!exists) return FALSE;
    /* Use that printer setup on this machine. */
    SetRegistryBinary(CUR, &setupVal,
        "HKEY_LOCAL_MACHINE\\System\\CurrentControlSet\\Control\\"
        "Print\\Printers\\HP LaserJet 4\\Default DevMode");
    /* Notify end user of required action. */
    NotifyUser(CUR,
        "Your printer setup has been updated to correct the "
        "'Document too complex' problem. Please try printing "
        "the document again.");
    return TRUE;
}
```

Listing 3: Solution code.

The customer feedback has been very positive. None of the customers have reported any problems or performance degradation with the software, and the software has proven its value in identifying major issues at every single one of the sites. The system administrator at one site was pleased to report that the logs have provided information on problems that would otherwise go unreported. At another site, the logs indicated a serious problem that was preventing the virus scanner from running, which had gone undetected for months!

The feedback from customer sites has also proved very valuable to our development team. The client checks every error return and uses its own reporting mechanism to log unexpected results. This process has unearthed several non-fatal, but potentially serious, bugs that were not found during testing. In early versions of the client, the problem resolution capabilities have been limited, so every event was escalated through the reporting mechanism. These reports have guided us in our choice of scrips to implement for the initial database, so we are confident that we are populating the database with the most useful scrips. One reassuring "reality check" is the observation that the problems that occur at all ten customer sites are remarkably similar, even though the businesses themselves are considerably different.

Open Source Development

The scrip database lends itself very naturally to an open source development methodology. The initial commitment to the development of a scrip is small

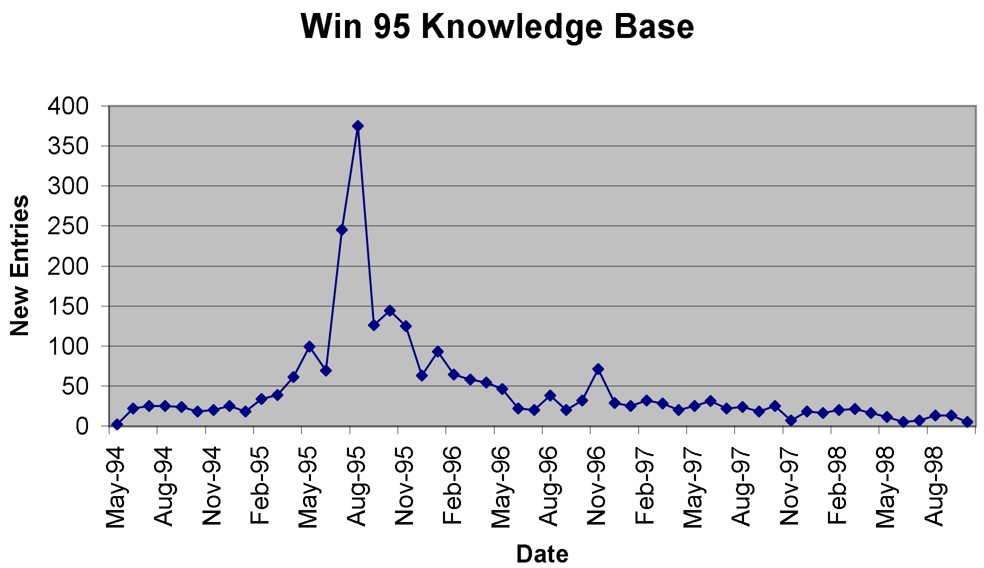

Figure 5: New Problem Incidence in Windows 95.

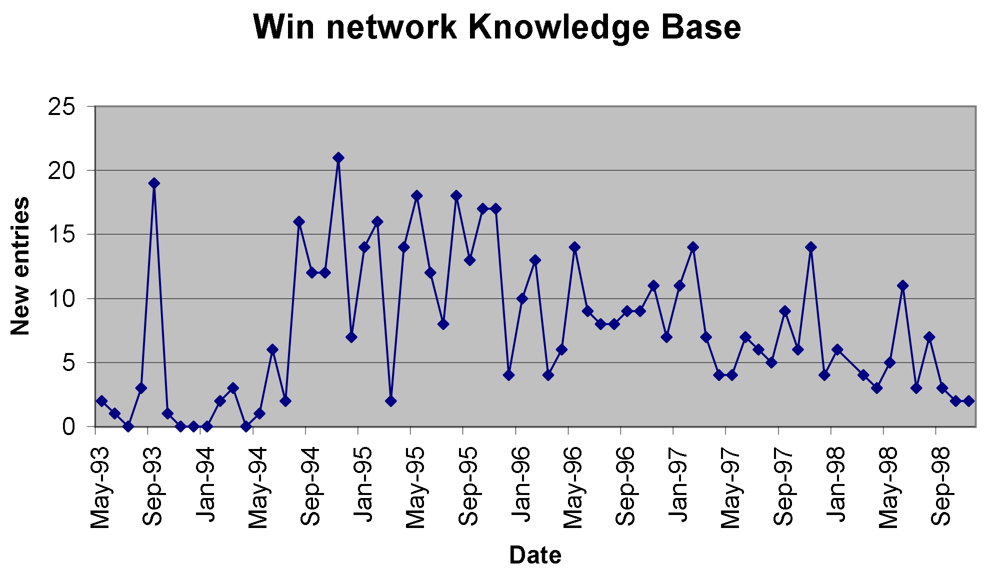

Figure 6: New Problem Incidence in Windows Networking.

since it only takes a few days, and the payback is immediate since it solves a persistent and annoying problem in a permanent way. Scrips are likely to be useful at more than one site, so the code that goes back into the open source pool will be immediately beneficial to the user community at large. This approach also helps to populate the database with the most important scrips first, since the motivation will be highest to solve the most severe problems. The open source methodology also adds a measure of security, since open review of the scrips will deter any malicious activity. A more subtle benefit arises from the fact that scrip development has several distinct phases, including problem resolution, solution definition, coding, and testing. These phases require different skills and can be completed most efficiently in a cooperative group environment. For all these reasons, we are using an open source methodology for scrip development.

Future Directions

Although this system is primarily focused on reducing system administration headaches by streamlining end user support, it lends itself naturally to several other uses that can also make an administrator's life easier. As previously described, we have written scrips both for bugs and for usability issues that are frequently encountered by end users and are confusing enough to generate calls for help. In addition, we have written scrips to automate common administration and maintenance functions that help to keep a properly functioning system running smoothly. We also see great utility in using scrips to distribute security patches, since the conditions for their use and the procedures for their application are often complex enough to warrant code for their implementation.

The technology is flexible and useful enough to apply to the next generation of "pervasive" computing devices. As the load on system administrators expands to include support of mobile and other embedded computing platforms, the need for automated support is only going to become more critical. The lightweight, standards-based client is ideally suited for these environments as well as the applications that support them.

Summary

The HandsFree Networks support automation tool described in this paper relieves system administrators of the burden of dealing with mundane issues by automating the resolution of many common recurring problems. It uses a standards-based extensible architecture to provide a database of solutions that is applied without manual intervention. An open source development methodology for the database and a built-in incremental database update ensures that the solutions will keep up with the introduction of new products and the corresponding appearance of new issues.

Author Information

Allan Miller received a B.S. in Electrical Engineering from Purdue University in 1979 and an M.S. in Computer Science from Stanford University in 1982. While on his way to a Ph.D. in Computer Science from Stanford, he left to start CASE Technology in 1983, to create and sell electronic CAD software. After CASE was bought by Teradyne in 1987, he left in 1993 to start Virtual Music Entertainment, providing a unique technology that allows non-musicians to enjoy playing music (the Virtual Music software is featured on Aerosmith's 1997 "Nine Lives" album). Allan likes to bicycle and play the piano, and he plays drums in the "well-known" Palo Alto band, The Wizards. He is active in town politics, serving on the Zoning Board of his Hollis, New Hampshire home town.

Alex Donnini has more than 20 years of experience in the information technology industry. During the past ten years he has acquired direct, extensive knowledge of the information technology challenges faced by small businesses and of their technical support needs. This led him, together with Allan Miller, to found HandsFree Networks. Their goal is simple: deliver the first automated support system initially targeting workgroup and departmental information systems. Throughout his career, Alex has specialized in getting products or companies off the ground. HandsFree Networks is his third start-up, the second one that he has co-founded. He received an Honors' B.S. in theoretical statistics from the University of Western Ontario in 1977, and a Master's degree in Business Administration from Harvard University in 1981.

References

[All99] M. Allimadi, "Companies Deploy Multi-Networked Call Centers to Deliver Efficient Customer Service," Call Center, http://www.callcentermagazine.com/article/CCM20000427S0016, June 1999.

[Bra98] T. Bray, J. Paoli, and C. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0," http://www.w3.org/TR/1998/REC-xml-19980210, Feb. 1998.

[CAS87] CASE Technology, Inc., *The CASE Design System,* marketing literature, 1987.

[Che99] A. Chen, "Sinking support costs," *eWEEK*, http://www.zdnet.com/eweek/stories/general/0,11011, 409655,00.html, July 1999.

[Che99a] A. Cheng, "Resurrected Technology for the Web: Expert Bites; AI Systems in digestible chunks," *Software911 white paper*, register at http://www.software911.com/form/default.asp, 1999.

[Com97] D. Comer and D. Stevens, *Internetworking with TCP/IP, Volume III: Client-Server Programming and Applications, Windows Sockets Version*, Prentice-Hall, http://www.cs.purdue.edu/homes/comer/tcpip3w.cont.html, 1997.

[CWS99] California Child Welfare Services, *1999 Year in Review*, http://www.hwcws.cahwnet.gov/Bulletins%20General/year_inreview.htm , 1999.

[DeK99] J. DeKeles, "Terror in the Land of Tech Support," ZDNet AnchorDesk, http://www.zdnet.com/anchordesk/story/story_3893.html , Sept. 1999.

[Etc98] J. Etchison, "Paradigm Schmaradigm," *IT Support News*, Viewpoint, http://www.itsupportnews.com/archives/9807_html/9807view.htm , July 1998.

[Fre96] A. Freier, P. Karlton, and P. Kocher, "The SSL Protocol, Version 3.0," *Internet Draft Memo*, http://home.netscape.com/eng/ssl3/draft302.txt , Nov. 1996.

[Gia99] G. Gianforte, "Eight Secrets for Successful E-Service," *Supportindustry.com Newsletter*, http://www.supportindustry.com/newsletter/110299.htm , Nov. 1999.

[Gil00] K. Gilhooly, "Certifying the help desk," *IT Support News,* http://www.itsupportnews.com/july2000/depts/gn/topstory2.htm , July 2000.

[Hon00] Honeywell, *Help Desk Services,* http://www.iac.honeywell.com/services/its/call_mgmt/help_desk.htm , 2000.

[IHS00] IHS Helpdesk Service, *Service Level and Metric Statistics*, http://www.ihshelpdesk.com/ , June 2000.

[ITS99] Editorial staff. "Problem Resolution at Work." *IT Support News*, http://www.itsupportnews.com/archives/9906_html/9906probres.htm , June 1999.

[Lan00] M. Lane, "Why does knowledge management matter?" *IT Support News*, Viewpoint, http://www.itsupportnews.com/feb2000/depts/si/sistory1.htm , Feb. 2000.

[Loc00] Lockheed Martin Services, Inc., *Help Desk Solutions Center*, http://www.lmsi-nw.com/it/helpdesk.html , 2000.

[Mic00] Microsoft Product Support Services, *Microsoft Knowledge Base*, http://search.support.microsoft.com/kb/ , June 2000.

[NCS98] NCSA HTTPd Development Team, *The Common Gateway Interface,* http://hoohoo.ncsa.uiuc.edu/cgi/ , Jan. 1998.

[Rea99] B. Read, "Outsourcing a Variety of Support Functions," *Call Center,* http://www.callcentermagazine.com/article/CCM20000503S0007 , July 1999.

[RSA00] RSA Security, *Call Handling and Escalation Process,* http://www.rsasecurity.com/support/techsup/escproc.html , May 2000.

[Sch00] Schlumberger, *GeoQuest Customer Support*, http://www.geoquest.com/pub/support/RGS/ Houston , 2000.

[Sla00] D. Slater, "Call Center Management," *CIO Magazine*, http://www.cio.com/archive/040100_numbers.html , Apr. 2000.

[Smi97] G. Smith, "Support for all," *ComputerScope,* http://www.techcentral.ie/cgi-bin/SiteWrapper.pl?template=/magazines/ComputerScope/article_template.html&target=/magazines/ComputerScope/1997/Mar/Features-0.html , Mar. 1997.

[Ste99] T. Steinert-Threlkeld, "Chatterbot: New answers from customer service," *Internet Business*, http://www.zdnet.com/zdnn/stories/news/0,4586,2188774,00.html , Jan. 1999.

[Sum98] J. Summa, "How to Build a Knowledgebase You Can Use!" *Customer Support Management,* http://www.customersupportmgmt.com/back/nov-dec98/know.html , Dec. 1998.

[Sup00] "E.SURVEY," *Supportindustry.com newsletter,* http://www.supportindustry.com/newsletter/032800.htm , Mar. 2000.

[Ter00] Teradyne, Product: VICTORY Boundary-Scan Software, http://www.teradyne.com/prods/cbt/products/pVICT/pVICT.html , 2000.

[Tol92] L. Tolan, "Managing the High Cost of Distributed Computing," Simon Fraser University Computing Services, http://www.sfu.ca/~lionel/Manage_Cost.html , Dec. 1992.

[Vir97] Virtual Music Entertainment, Inc, Products, http://www.virtualmusic.com/products/ , 1997.

[Wal00] L. Wall, T. Christiansen, and J. Orwant, *Programming Perl, 3rd Edition,* O'Reilly, http://www.oreilly.com/catalog/pperl3/ , July 2000.

[Yah00] Yahoo, Yahoo! search engine, http://www.yahoo.com .