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

RUST: Managing Problem Reports and To-Do Lists

Craig Ruefenacht
University of Utah

RUST: Managing Problem Reports and To-Do Lists

Craig Ruefenacht – University of Utah

ABSTRACT

Organizing and managing to-do lists and user reported problems efficiently are important tasks for system administrators. The overhead of organizing them involves time. As the to-do list and problem reports increase, so does the time required to organize and maintain them. Without an effective method of managing these lists, important issues may be ignored and forgotten.

This paper describes our experiences in evaluating and implementing problem and work tracking software. Initially, we examined and tested currently available software. We found, however, that they poorly addressed some problems facing our computing facility. We therefore designed and implemented a ticket system that would best fit the needs of our network.

Defining the Problem

E-mail is a tool commonly used to report problems on a network. In addition, problems are also reported in the hallway, on sticky notes stuck to doors, and over the phone. Without an effective way of managing these problem reports, some of the important problems go unaddressed for long periods of time. The longer a request goes without being looked at, the higher the chance for it to be forgotten. This creates an unhealthy atmosphere for users, staff, and researchers.

The experience we have had is similar to other sites, which are discussed in [1], [2], [3], and [5]. Users of our network did not understand why it took a long time for us to address user reported problems. Some users expressed concern about this latency of problem resolution. Users also expected individual staff members to remember problems discussed in the hall and over the phone. They were unaware of the number of problems and issues facing our facility staff. Besides taking care of user reported problems, our staff maintains and upgrades various software packages, implements changes that reflect current and future growth estimates, installs new hardware and computer labs, and takes care of the administrative items needed to keep a UNIX network running. Without having a system for managing these issues, some of them are overlooked and sometimes end up in a "black hole".

Site Information

The University of Utah College of Engineering and Computer Science computing facility is a heterogeneous network of more than six hundred workstations and fifty servers, running twenty-four different operating systems. These include SunOS (v 4.1.2, 4.1.3, & 4.1.4), Solaris (v 2.3, 2.4, & 2.5), HP-UX (v 9.03, 9.05, & 10.0), Irix (v 5.3, 6.1, & 6.2), AIX (v 3.2 & 4.1), BSD (v 4.3, 4.4, NetBSD, and our own homegrown version), Ultrix v4.3, Linux, Microsoft Windows NT (v 3.51 & 4.0), Microsoft Windows95, and Macintosh OS. Our network is used by more than forty-five hundred users for tasks ranging from classwork to faculty research. Both NFS and AFS filesystems support our network, providing services and close to a terabyte of disk storage. We have five Fast Ethernets, four FDDI Token Rings, and more than thirty-five ethernet segments connecting the various pieces of our network together.

Many problems arise with this kind of network. Over the past ten months, more than seventy-two hundred problem reports have been submitted to our facility staff, the majority coming from novice users. In addition, more than two hundred work requests have been logged and processed. During the beginning weeks of an academic quarter, the average number of reports received per day can reach seventy-five or more. The facility staff, composed of nine full time system administrators, three full time and two part time hardware technicians, and eight part-time student operators, must handle all these problem reports.

Things Needed to Change

The problem reporting system we were using a year ago was inefficient. Our facility problem reporting system was broken down into five e-mail mailing lists. Each was devoted to a subset of all problems being reported. When users sent e-mail reporting a problem to a mailing list, it was forwarded to the staff members on the particular mailing list. The rule of thumb was that the first staff member to decide to take care of the problem report would be responsible for sending a reply back to the user. They would also be responsible for informing the other staff members that they were working on the problem.

A problem with this setup was that our staff members did not know who was working on which problem reports. Sometimes this caused overlap where two people would work on the same problem at roughly the same time. It was usually after the fact that both staff members found out about the overlap.

Another problem was having pieces of e-mail from users get lost in the stacks of e-mail from other users. Trying to manage hundreds of messages in your inbox is an art that even experienced system administrators can struggle with. The embarrassment comes a couple of weeks later when the user inquires further about the problem, only to find out that somehow no one knew the problem existed.

These problems were supporting an unhealthy computing environment. It was not providing good rapport between us and the user community. To help combat some of these issues, we began to evaluate ticket systems, which are designed to alleviate the problems we were experiencing.

Ticket Systems

Ticket systems are designed to manage a list of items for a group of people. There are several such systems available for Internet use. They range from complex commercial software that uses relational databases and efficient search engines to simple public domain systems designed around text-based or dbm format databases.

There are two types of ticket systems used in system administration today. One type, commonly known as a trouble ticket system, is used to organize problems reported by the user community of a network. This type of system assumes that each reported problem has a specific solution even if the problem is recurring. Once a problem has been resolved the first time, information about it is stored and can be recalled if the problem manifests itself at a future date.

The other type of ticket system, commonly known as a work tracking system, is used to help maintain to-do and work lists. It is also used to keep track of work being planned or the work currently being done by the system administrators. Work tracking systems are used by system administrators and do not involve the general user community.

Both types of systems work on the same principle. Each request or to-do item, also called a ''ticket,'' is assigned a ''ticket'' number. These ticket numbers are used to keep track of each request or to-do item. Information about each ticket is stored in a database and can be recalled for future reference.

The Search For a Solution

Several issues need to be addressed to formulate a solution. How can a software system be best implemented to keep track of the large volume of e-mail that report problems? Does it require much overhead to maintain the system and can it function on its own? Is it easy for users to use and can they use it to get feedback about reported problems? Is it secure enough to allow users read access to the database of tickets without inadvertently allowing modification? Can the system be configured so that it can also handle the to-do and work lists of our staff? Would it be flexible enough that other programs could be interfaced with it at some point, such as financial accounting software?

Existing Systems

We examined several public domain software packages, which are listed in Appendix A. We decided to stay with public domain because our budget does not have much room for the costs of commercial systems. There are a few systems available that have most of the features and abilities we need. But they were financially infeasible for us to implement.

Existing Features

In the public domain software, there were many features common to all the systems we examined. Each system had an interface that used e-mail to create and add information to tickets. This was an important feature for us because most of our problem reporting is done via e-mail.

Some systems are based on specific mail implementations that limit how they can be utilized. The NEARnet Trouble Ticket System, for example, is based on MMDF and currently does not support sendmail implementations. Another system called Queue-mh is a set of scripts based on the MH mail system. This system is discussed in [5]. To use most of the system, one would have to be using MH. This posed a problem for us because our staff uses a wide range of mail programs, including GNU rmail, pine, elm, MH, zmail, and netscape. In order for our staff to use it effectively, we would all have to convert over to MH.

All the systems we evaluated also implemented a database to store information on each ticket. While a couple of the systems use a relational database format, most use plain ASCII text files, with each ticket being stored in its own file. The ASCII databases are easy to maintain and troubleshoot, and can be looked at with standard UNIX utilities. The use of a simple database structure appealed to us because it does not require special interfaces to access stored data.

User feedback implementations are also standard features among the systems under examination. The most common method provided is a user interface. These interfaces allow users to gather information about problem reports submitted by the user. Some systems also send out automatically generated e-mail messages to users who have submitted problem reports.

After looking through what was available, we decided to install a system called REQ, which was written by Remy Evard at Northeastern University in Boston, Massachusetts, and is discussed in [2]. This system was easy to install. It is written to work with sendmail, which is what our mail system is based on . REQ is written in Perl with several C setgid wrappers around the Perl code.

Missing Features

Once REQ was installed, it became apparent that we needed features which REQ did not provide. Most of these features were not apparent to us until we implemented the system.

The key feature missing was support for multiple mailing lists. As mentioned earlier, we had five mailing lists used for problem reporting. We also had a few other lists that could use a management system. In order for us to maintain several mailing lists on REQ, we had to install multiple copies of REQ, one for each mailing list. This setup became burdensome and was not very practical.

Another problem we ran into was that there were too many C wrappers for REQ. When we installed the system for our five mailing lists, we had a set of seven binaries for each mailing list for each architecture. That comes out to be thirty-five binaries. Multiply that by nine architectures in which we installed the system onto, and you come out with three hundred and fifteen binaries. This setup was more complicated than we wanted.

Making It Work

To make REQ work for us, we had to do several modifications to the system. We removed all C wrappers to make maintenance easier. Removing these wrappers also meant that all our staff members needed to have read/write access to the database of tickets so they could add comments and do other operations on the tickets. For a ‘‘closed’’ system in which only our staff was using REQ, this setup worked well.

The biggest modification that needed to be done was to enable one copy of REQ to manage several mailing lists. This involved several changes to the underlying system. A good understanding of the low level structure of REQ was required to do this.

After these modifications were in place, we were able to implement REQ using only one copy to manage our five mailing lists. Other sites using REQ were also trying to implement it for multiple mailing lists. Because of this need, we made our modified version of REQ available to the Internet community. Because our additions had to use much of the existing low level structure, they were not the most efficient. Although our alterations to the system worked, other issues began to surface which would require more modifications.

Along Came RUST

With the REQ system needing more modifications to resolve some issues that came up, the system was becoming more of a hack and less of an efficient system for managing e-mail. Development of REQ was starting to diminish. The other systems we examined also did not have much development currently going on. The hacks we were adding to REQ continued to be limited to the existing code, and it was becoming necessary to rewrite major parts of it to improve performance for our setup.

Using the results of our examination of ticket systems and our testing of REQ, we decided to formulate our own solution to the problems we were facing. Under the guidance of our facility director, we began to design our own ticket tracking system. The system, called the **R**equests, **U**sers, and **S**ys-admin **T**o-do Ticket System, (**RUST**) is centered around redesigning and improving what other ticket systems have. RUST also has added functionality and features to broaden its application beyond just problem reporting.

The Nitty Gritty

How RUST Works

The basic operations of the RUST System are common among several existing ticket systems. The main tool that RUST employs as its interface is e-mail, based on an underlying sendmail implementation. Tickets are created in the RUST System from e-mail sent to it. Information can be added to existing tickets by sending e-mail to the RUST System with a special ‘‘tag’’ in the subject line.

E-mail is not the only interface to RUST. Operations can also be done via text and GUI based interfaces. This includes creating new tickets, sending e-mail in response to a specified ticket, and updating information. Our staff can also work on the tickets by either e-mail or interactive interfaces. This provides our staff the freedom to use an interface of their choosing.

Multiple Mailing Lists

The key feature of RUST is its ability to manage multiple mailing lists with only one copy of the system installed. This eliminates the need to install a separate copy of the system for each mailing list.

At the top level, RUST handles multiple mailing lists in much the same way as the Majordomo Mailing List Manager as discussed in [4]. Each mailing list is unique and has its own configurations. RUST maintains each mailing list independently of any other mailing list, and provides a mail interface, much like Majordomo, enabling users to request information about specified tickets.

To-do Lists and Work Tracking

Keeping track of problem reports is only part of the solution. Besides taking care of problems, we also do other tasks. Part of the added functionality of the RUST System is the ability to keep track of what our staff members are working on. This includes ongoing projects and to-do items that are in either their planning or implementation stages.

Other problem tracking software can be used for work as well as problem tracking, but those systems make no distinction between the two. A work ticket looks the same as a problem ticket. RUST resolves this problem by defining two types of tickets; user reported problems and work tickets. RUST can also store financial accounting information and the total time spent on each ticket.

The Path For E-mail

A major concern we were facing as a computing facility staff was the path of e-mail between users and staff members. Users would often send mail to a specific staff member, asking that this or that problem be addressed quickly. Because the e-mail went to only one staff member, the rest of our staff would have no knowledge of the problem. Another concern we had was the situation where a user would send more e-mail concerning a problem to the staff member taking care of the problem, without carbon copying the rest of the staff members.

Removing these variables of sending e-mail was something we felt would be a great help to managing it. The optimum solution would be to have both the users and staff members use the same system for sending e-mail. While expecting all users of the local network to use the same "program" to send mail was not feasible, having the users and staff members both send mail only to one place seemed reasonable.

The RUST System implements a topology of e-mail where our staff members and user community both send mail to one place. To accommodate this, the RUST System keeps track of several pieces of information about each ticket. One of these is a list of e-mail addresses for each ticket, containing all addresses that have ever gotten any e-mail concerning the specified ticket. When e-mail concerning an existing ticket is received, RUST ensures that each address listed for the specified ticket receives a copy of the e-mail.

This topology for e-mail does not remove the inter-staff e-mail that our staff members send each other. We continue to use inter-staff e-mail for discussing various issues and concerns. It is only e-mail that goes between our staff members and the users that goes through the RUST System.

Local User Access

In existing systems, user interfaces are dependent upon having the ticket database files mounted on the machine on which the user uses the interface. To provide some protection, interfaces are either setgid or setuid to a group or user who can read the database.

For better protection, RUST implements a server/client framework using TCP UNIX sockets. This framework is not dependent upon having the database files mounted on all machines on which a user may run the client interface. The client interface makes a TCP connection to the server to retrieve the requested information. The server runs on the machine that the RUST System files are located. All RUST System files are owned by a pseudo user, and the server runs as this pseudo user. To add maximum protection, the server does not modify any files.

Accessibility For Remote Users

In existing software, there are not many provisions made for remote users to get feedback on problem reports they have submitted. The remote user usually does not have any access to the local network except through e-mail and World Wide Web services.

For the RUST System, we are designing a mechanism for remote users to receive feedback and information regarding their submitted problem reports. A remote user will be able to send e-mail to the RUST System on the local network to request information. The RUST System will process the request and serve the requested information. This mechanism is similar to the Majordomo Mailing List Manager, where Majordomo automatically processes requests sent to it via e-mail [4].

We are currently assessing issues concerning limiting remote access in developing a safe implementation. A key issue is how to prevent remote users from finding possible break-in points of the local network by exploiting its insecurities. Remote users should only be able to request information about reports that they themselves have submitted.

Platform Independency and Protection

With a heterogeneous network, achieving as much platform independency as possible is essential. We decided to ensure that all executable code needed for RUST is platform independent. To achieve this, we decided to write RUST in Perl. Doing this gave us the greatest flexibility across platforms and greatly reduced the overhead of maintaining the system.

During the implementation stage of RUST, there were some problems in having everything written in a scripting language. The UNIX operating system environment does not provide a safe mechanism to run programs written in scripting languages as setgid or setuid. Not having the ability to do this meant that any files that staff members needed to have access to would have to be readable by themselves, with a few having to be writable.

If the RUST System were being used in a "closed" environment, where all staff members could be trusted to have read/write access to the RUST System files and database, there would be no problem. However, the RUST System has the potential to be used by more groups than just our facility staff. When expanding the usage of RUST to general user applications, such as mailing lists dealing with software bug reporting, which may have nothing to do with our facility staff, database protection becomes an issue. Putting the users who are on software bug reporting mailing lists and our staff in the same group and enable all of them the same read/write access to all RUST System database files is not wanted.

There are several ways to implement protection to resolve this issue. The method that RUST will use on a permanent basis has not been determined, as configurations of various types are being looked into. While having everything written in a scripting language is optimal, sometimes it is not feasible.

Currently, RUST employs a single binary wrapper for each platform that is setgid to a UNIX group that has read/write access to all RUST System database files. The wrapper in turn calls interfaces to the RUST System, which are written in Perl. These scripts do internal user authentication checks to ensure that the user running the interface only does modifications to RUST System files that he or she should be able to.

Modularity

One idea that gained support early on in designing and implementing RUST was the idea of modularity. While looking at other ticket systems, we found that most of them were part of the interfaces to the system. If we wanted to redesign the interfaces to the system, it was not a trivial task. Changing how the interface worked invariably affected the rest of the system in some way.

By employing modularity in the design of RUST we were able to implement a system that was interface independent. The support that Perl provides for module functions and files provided a framework in which this could be done. The core of the RUST System is written as Perl modules, being divided into several module files.

Using modularity enabled us to design simple top-level interfaces. They do nothing more than gather information interactively from a person and then make the appropriate calls to module functions. It becomes a trivial task to modify the interfaces to suit the tastes and needs of our staff.

Designing interfaces is not limited in what can be done. Interfaces of any type can be designed without having to know the low level guts of the RUST System. This also provides a framework in which other software can interface with the RUST System with little effort. Such software packages may include systems that automatically detect problems before they become big problems, accounting software that keeps track of which staff members are doing what, and search engines designed to search through requests for information.

Database Design

There are a variety of database frameworks that were considered for the RUST System database structure. It was decided to implement the database as a set of ASCII text files. Each ticket in the RUST System is associated with its own file. Each file contains the contents of the original ticket. Whenever new information is received, it is appended to the file.

The structure of the database is broken down further to contain two files for each ticket.

Header File

The first of the two files is called a ‘‘header’’ file, which contains the administrative information about each ticket. Such information includes the user who created the ticket, the staff member who is working on the ticket, a list of e-mail addresses who should get a copy of e-mail concerning the ticket, financial accounting information, and what kind of priority the ticket has.

Ticket Logfile

The other file is a log kept for each ticket. This file is in RFC822 format, so any mail program that supports RFC822 mailfiles can read and understand the ticket logfile. This is where all e-mail concerning the ticket, comments added by staff members and users, and operation log messages are stored. New data is appended to the end of the file, so it can also be used as a chronological record of progress.

```
usage: check-q [options]

    options you can specify are these:
          --all
          --category <user|work>
          --help
          --owner <owner>
          --printer <printer_name>
          --priority <priority>
          --queue <queue_name>
          --show <num>
          --status <status>
          --subject <subject>
          --type <active|resolved|archived>

<queue_name> can be one of:
      opers unixsys dsl accounts printers modems rust-bugs
```

Figure 1: Usage for check-q

Example Interfaces

User Interfaces

The *check-q* program is a text based interface for use by the general user to retrieve information about problem reports they have submitted. Its usage is summarized in Figure 1. A sample use of *check-q* is shown in Figure 2. It displays a one-line sumary of each ticket that matches the query. Another use of *check-q* is to request the entire logfile for a specified ticket. This can be done as shown in Figure 3.

Administrator Interfaces

In addition to the *check-q* interface, our staff has another one called *action*. It is used to perform operations on tickets. Its usage is summarized in Figure 4. For example, if one of our staff members wants to assign a ticket to a fellow staff member, they issue the following command:

```
action --queue rust-bugs --assign \
        johndoe --num 10
```

The above command will assign ticket number 10 in the rust-bugs RUST list to the user johndoe.

```
command -> check-q --queue rust-bugs

Date: Wed Jul 31  0:35:52 1996
Query Filters:
        Queue: rust-bugs
         Type: active
         User: ruefenac
         Owner:
         Status:
         Priority:
         Category:
         Subject:

RUST#  Owner    Priority  Status Subject
--------------------------------------------------------------
  4    ruefenac normal    open   RUST doesn't parse mail right
```

Figure 2: Using check-q to query

```
usage: action [options]

    options you can specify are these:
           --add_time <hh:mm> (add time to work_hours)
           --append user
           --assign <user>
           --comment
           --create
           --kill
           --merge --num1 <num> --num2 <num> (puts num1 into num2)
           --queue <queue_name>
           --num <request_number>
           --set
                category
                eta
                prio
                status
                subject
                user
                accounting
           --send <user>
           --show <num>
           --value <string>
```

Figure 4: Usage for action

```
command -> check-q --queue rust-bugs --show 4
=================================
TICKET INFORMATION
=================================
Ticket Queue: rust-bugs
Ticket Number: 4
Ticket Creation: Tue Jun 25 20:32:13 MDT 1996 (835756336)
Ticket Updated: Thu Jun 27 20:10:58 MDT 1996 (835927860)
Ticket Notification:
Ticket Category: work
Ticket Subject: RUST doesn't parse mail right
Ticket Type: active
Ticket User: ruefenac@cs.utah.edu
Ticket Owner:ruefenac
Ticket Status: open
Ticket Priority: normal
Ticket ETA:
========================================
TICKET LOGFILE
========================================
From ruefenac@cs.utah.edu  Tue Jun 25 20:32:12 1996
Date: Tue, 25 Jun 1996 21:32:00 -0500
To: rust-bugs@cs.utah.edu
Subject: RUST doesn't parse mail right

This needs to be fixed...

RUST isn't getting the "From" header right. Take a
look at the debug files and see where the problem is.

-Craig
==========================================
Date of creation: Tue Jun 25 20:32:13 MDT 1996

From RUST Thu Aug  8 17:24:28 1996 (839546671)
Subject: Comments added

I'm looking at this problem and will have it fixed shortly.

-Craig
=========================================
Date comments were added: Thu Aug  8 17:24:28 1996

From ruefenac@emily.cs.utah.edu  Thu Jun 27 20:10:58 1996
Date: Thu, 27 Jun 1996 20:10:55 -0600
X-Mailer: Z-Mail (3.2.0 06sep94)
To: rust-bugs@cs.utah.edu
Subject: Re: [rust-bugs #4] RUST doesn't parse mail right

%> message from ruefenac
%>
%> This needs to be fixed...
%>
%> RUST isn't getting the "From" header right. Take a
%> look at the debug files and see where the problem is

Fixed...
-----------------------------------------------------
Craig Ruefenacht                 ruefenac@cs.utah.edu
-----------------------------------------------------
=========================================
Date mail was appended: Thu Jun 27 20:10:58 MDT 1996
```

Figure 3: Using check-q to display entire logfile

GUI Interfaces

GUI interfaces are also provided for both our staff and users that do the same thing as the *check-q* and *action* interfaces. They are written in Perl using the Tk modules to provide the interface to X.

The Results

The results of implementing our own ticket system resolved many issues we were facing and are summarized below:

- All e-mail coming from users which report problems are organized into one system, enabling us to better manage them.
- Our staff knows which problem reports each staff member is working on. Its no longer a guessing game as to who is doing what
- Work and to-do items are managed in the same system as the user reported problems. The work and to-do items appear side by side with the problem reports, enabling our staff to see the amount of work that needs to be done.
- Interface independency was achieved. This allows interfaces to be written without having to modify the low level structure of the system.
- Users are able to see the amount of work that our staff needs to do. This leads to better user understanding as to why the latency of problem resolution is sometimes large.
- The rule of thumb has become "if it isn't in the RUST System, then it isn't a problem". This has removed the problems reported in the hallway and via personal e-mail. The idea takes a little getting used to, as you can't expect the change to be accepted nor implemented overnight.
- Users no longer have to send e-mail to our staff to find out about the status of their reported problems. Interfaces allow the user to obtain this information on-line.

Other benefits resulted from the side affects of this project:

- Other groups at our site saw the RUST System as being useful for software debugging and other mailing lists. We are currently adding six mailing lists which have nothing to do with our facility staff to the RUST System.
- We were able to discontinue the practice of sending a copy of all user e-mail to each staff member on a particular mailing list. RUST processes the mail, logs it, and that is where the e-mail stops. The result was reducing by a large factor the mail that was being sent to staff members.

Future Work

RUST System development is an ongoing project here at the University of Utah. Each component of the system was carefully designed before it was implemented to ensure that the most efficient methods were used. There will always be development on the RUST System as the needs of our network change.

World Wide Web

Currently, there is no World Wide Web user interface to the RUST System. An interface of this type would not only be useful for local network users, but would also enable remote users to get feedback on their requests. There are many concerns with a World Wide Web interface. The key issue is user authentication so that access can be limited based upon the user requesting information.

Mail Server

Several issues are involved with the mail server. A user mail server is currently under development and different configurations are being tested. Because problem reports and to-do items can provide outside hackers with valuable information, the mail server implementation will include support for data encryption and authentication.

Availability

The RUST System has been designed to work on other sites besides our own. Its application is general enough that it can be used for problem reporting, to-do list and work management, bug reporting, and other similar applications.

Licensing for the RUST System is provided by the GNU Public License and may be freely distributed. It is available via anonymous ftp to ftp.cs.utah.edu in the /pub/rust directory. This is mirrored at the University of Minnesota, ftp.cs.umn.edu, in the /pub/rust directory. There is also a Majordomo mailing list devoted to general discussion of the RUST System. The address of this list is *rust@cs.utah.edu*. Anyone can subscribe to this list by sending e-mail to *majordomo@cs.utah.edu* with *subscribe rust* in the body of the e-mail.

Documentation is provided with RUST on all module functions that are available for use in interfaces. Examples are also provided for those who embark on writing their own interfaces. As with text-based interfaces, the same ideas can be applied toward GUI interface development.

Acknowledgments

The development of the RUST system has primarily been done here at the University of Utah. The facility staff has been very helpful in the design and implementation stages of RUST. Aleksandra Kuswik, Kiri Wagstaff, and Michael Chang have also contributed to this project. The information provided from these people have been significant to the RUST System project.

A significant contribution for GUI interfacing has been provided by Shawn Amundson and the Computer Science Department of the University of Minnesota. Other system administrators from the Internet

have also submitted suggestions for making RUST a more robust system. This has enabled the ideas of others to be a part of our design.

Author Information

Craig R. Ruefenacht is a full time student attending the University of Utah where he is working on a bachelor's degree in Computer Science. Craig works for the Computer Science computer facility staff as a part-time operator and systems programmer. His current interest is improving the environment in which system administrators work. You can reach him via US Mail at the University of Utah Department of Computer Science, MEB 3190, Salt Lake City, UT 84112. His e-mail address is ruefenac@cs.utah.edu.

References

[1] Elizabeth D. Zwicky, "Getting More Work Out of Work Tracking Systems", *Proceedings of the 8th Large Installation Site Administration Conference*, pp 105-110, San Diego, CA 1994

[2] Remy Evard, "Managing the Ever-Growing To Do List", *Proceedings of the 8th Large Installation Site Administration Conference*, pp 111-116, San Diego, CA 1994

[3] David Koblas & Paul M. Moriarty, "PITS: A Request Management System", *Proceedings of the 6th Large Installation Site Administration Conference*, pp 197-202, Long Beach, CA, 1992.

[4] Brent Chapman, "Majordomo: How I Manage 17 Mailing Lists Without Answering "-request" Mail", *Proceedings of the 6th Large Installation Site Administration Conference*, pp 135-143, Long Beach CA, 1992

[5] Tinsley Galyean, Trent Hein, and Evi Nemeth, "Queue-MH: A Work Queue Management Package for > 3 Ring Circus", *Proceedings of the 4th Large Installation Site Administration Conference*, pp 93-96, Colorado Springs, CO, 1990.

Appendix A

This appendix lists the currently available public domain non-commercial ticket systems we examined. This list is provided in hopes that it may be useful to other sites.

- **GNATS** – Located at prep.ai.mit.edu:/pub/gnu/ as gnats-3.2.tar.gz. This is the GNU Problem Report Management System. It is designed for bug reports, but can be used for other applications such as trouble tracking.
- **NEARNET** – Located at ftp.near.net:/outgoing/ as nearnet-ticket-system-v1.3.tar.gz. This ticket system was written at BBN Systems and Technologies. It uses an Informix Relational Database, Embedded-SQL, and MMDF.
- **NETLOG** – Located at ftp.jvnc.net:/pub/GES-Other/ GES-Packages/netlog as netlog-tt-2.4. tar.Z. This system uses standard UNIX tools and an ASCII database structure to manipulate requests and logs
- **PTS** – Located at ftp.x.org:/contrib/applications/pts/ as pts-1.05.tar.gz. Written by Dean Collins, Chris Hyde, Luke Sheneman, Brian Goff and John Hunter, PTS is a system based on X11R5/R6 and the athena widgets.
- **Queue-MH** – Located at ftp.colorado.edu: /pub/sysadmin/utilities/ as queuemh.tar.Z. Written by Tinsley Galyean and Trent Hein, this system is a set of scripts designed to interface with the MH mail system on UNIX platforms.
- **REQ** – Located at ftp.ccs.neu.edu:/pub/sysadmin/ as req-1.2.7.tar.gz. This system was written by Remy Evard at Northeastern University. It is written mostly in Perl, using some C programs as wrappers. The database is ASCII based, and there is a tcl/tk and emacs interface for it.