A Highly Configurable Software Bug Tracking System

by

Calista Tait

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Feburary 1999

Author .... [illegible]
Department of Electrical Engineering and Computer Science
Feburary 3, 1999

Certified by .... [illegible]
James D. Bruce
Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Highly Configurable Software Bug Tracking System

by

Calista Tait

Submitted to the Department of Electrical Engineering and Computer Science
on Feburary 3, 1999, in partial fulfillment of the
requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and
Computer Science

Abstract

The design and partial implementation of a bug tracking system which is highly configurable with respect to user interface, information stored, manipulation of information, and client features are described. Additionally, the database used, the level of security, and other system maintenance concerns are discussed. Attention is given to methods which keep the design for such a system flexible, without losing accountability.

Thesis Supervisor: James D. Bruce
Title: Professor

Acknowledgments

Thanks to Ted Ts'o who served as my advisor. Thanks to Greg Hudson who authored the .zwgc.desc on which the thesis version of the same file was based. Thanks to John Hawkinson, Erik Nygren, and J.B. Tait for their proofreading efforts. Thanks to Lara Karbiner, Bill Sommerfeld, Ping Shuang, Eri Izawa, Greg Hudson, Jamie Morris, Sam Hartman, John Hawkinson and Mark Eichin for their input on the bug tracking systems they use. And lastly, thanks to all those who kept my spirits up.

Contents

List of Figures

Chapter 1

Introduction

1.1 What a Bug Tracking System Should Do

When maintaining any large program, or set of programs, there are always problems discovered by the users and the developers. Receiving, organizing, and assigning reports on these problems is important to the successful maintenance of any major piece of software. A "bug tracking" system attempts to perform this task.

There are a number of important considerations for the success of a bug tracking system. First, people using the software on which the bug reports are based, should find it easy and convenient to submit reports. This means that the interface for submitting bug reports should be simple and familiar. If it is not, reports will not get submitted and the system will be ineffective. Second, the interface for organizing the reports should be acceptable to the people who have to read, analyze, and respond to the reports. If the people responsible for fixing the software for which the bug reports were submitted will not use the tracking system, the system will, again, not serve the purpose for which it was designed. Third, the system should be flexible in how it deals with the information contained in the bug reports and the information that the people addressing the reports wish to add. This means it should be easy to add new fields, new methods of searching the database, and new ways of finding similar reports, so that the person fixing an error can have as much information as possible. Finally, the system itself must be easy to maintain. It should be comprised of modular code with good documentation and a logical design.

1.2 Common Problems

1.2.1 Scalability

A successful software system will be in production for a long time, and a single subsystem will be used for many packages of code. For this reason, a successful bug tracking system should be able to run correctly, and with consistent performance, even when there is a large number of reports. It is important to ensure that the reports are stored in such a way that they do not require an excessive amount of resources.

1.2.2 Reliability

Losing a report can have serious consequences. It might, for example, have contained a crucial clue to reproducing the bug, or it might have been a report regarding a security problem. For this reason, it is useful to have the option to implement a design that facilitates reliable storage of the reports, reliable searching, and regular backups.

1.2.3 Accommodating a Variety of Support Processes

With every software package, or with collections of software, there is, or should be, some sort of support and release process. Distributions are made, patches are suggested and reviewed, and many other considerations have to be accommodated. Once a bug is fixed, the new version is usually not instantly accepted into the main code repository. For this reason it is important that a bug tracking system allow for having bugs solved but not resolved, and must fit in with any reviewing and releasing procedure. Additionally, the bug tracking system should have the ability to retain the status of the bug as it goes through the other systems of support.

1.3 Other Bug Trackers

There are, to date, countless bug tracking programs, each written for a different purpose. Some are well-known and some are home-grown for use within a single company. I will highlight a few to provide a basis for this thesis.

1.3.1 netprob

Netprob is a bug tracking system which is client/server based, and has two user interfaces: a Motif graphic user interface (GUI), and text. It has abilities to classify, chain, and archive bug reports. It also has a facility for configuring the default selection of reports it will display when started, and the reports for which it will generate reminders on a weekly basis. Submission of bug reports is by email.

1.3.2 GNATS

GNATS is an email-based system with a relational database back-end. It has been used for multiple tasks, and there has been a web interface developed for it. It efficiently searches by field, and displays the collected reports. Its submission program enforces certain fields to contain valid values and it has abundant submission entry slots. All changes to reports are made with an editor interface.

1.3.3 jitterbug

Jitterbug is a web-based tracking system which receives reports either by email or in web form. It allows users to add notes, and it has categorization by authentic users. Authentication is dependent on the browser. *Jitterbug* has email notification, searches, personal configuration, and it keeps records of all changes to reports. Instead of a database, it uses flat, separate files to store the reports.

1.4 Case Studies

To better understand the real problems with bug tracking systems, I asked a group of computer scientists what features they liked or disliked in the production systems which they already used. A sufficient number of the people used *GNATS*, that section 1.4.2 is devoted to it.

1.4.1 A Collection of Comments on Bug Tracking Systems

Opinions were collected on the bug systems *Clarify, PV/BugWorks,*[1] *RAID, discuss,*[2] *netprob, PureDDTS, Vantive, Debian, QARadar*, and simple email.

The comments which were made can be summarized in the following categories:

- Platform dependence: A lot of these systems are only accessible from one platform, and in many of the cases, it was not a platform which the user was currently using.

- A variety of user interfaces: Many people either complimented their bug system for having many user interfaces, or scorned it for lack of adequate user interfaces. One of the most important user interfaces that respondents requested was a command-line interface. This allows for quick queries without starting up a bulky GUI. Text-based interfaces can also be used over any remote access, such as telnet, or even one as primitive as a text dialup connection, and it can be adapted by the blind or other handicapped users who cannot use a graphical interface or a mouse. Interfaces should also be interactive. Interfaces purely based on email, for example, do not give you realtime feedback for viewing and altering the reports.

- Buggy systems: If a system crashes frequently, loses or corrupts information, or does not consistently do what it should, people will not use it.

[1]SGI's home-grown system. It has many user interfaces and an SQL database.

[2]This is an MIT system which archives messages. It has a search tool, but other than that, nothing which allows for efficient sorting and viewing of bugs and their progress.

See also section 1.2.2.

- Abundant information: Clear and flexible information categorization facilitates ordering, searches, and accountability for the bugs and associated knowledge.
- Posting of new information: Since new information can be added to old reports, it is desirable to have a method to identify what information is new. Realtime updates, to either the software maintainers or a company's clients, are also important to keep everyone informed.
- Database segregation: It is sometimes useful to have multiple sets of information accessible using the same server/client pair. There should also exist a method for transfering information between two such databases.
- Multiple field searches: There are often times when one wishes to search a set of fields, rather than only one.
- Keyword searches: If there is a collection of words which are deemed representative of the bug report, searches are more likely to return relevant reports.
- Trustworthy and flexible security: There is no single correct method of security. Security methods also become obsolete over time. It is also important to be able to adjust which functions of the system are controlled by security.[3]
- Configurable work flow: It is important that there be as many, or as few states as needed to classify a bug. One application of the system might have more levels of the bug fixing process than another.
- Scalability: A system should perform well when the problem gets bigger.

  See section 1.2.1.

[3]The bug tracking system *netprob* had the problem that anyone who had permissions to close and archive bugs also had permissions to do much more drastic things. For example, if a user with archiving permissions were using the text client, they could kill the *netprob* server with the single character 'K'. Restarting the server was not as trivial.

Many of these points, (platform independence, user interfaces, categorization, posting new information, multiple field searches, keyword searches, trustworthy and flexible security) either already exist in my system as features, or can easily be added. When the "bug status" field is in place, it is trivial to add configurable work flow. A discussion of database segregation can be found in section 5.1.2 and of scalability in section 5.4.

Every system has software bugs; it depends upon the authors and the maintainers how well or how badly a system performs. My system, as I explain in the implementation chapter, is easy to maintain. I have done all I can to ensure that this aspect would be present in the system.

1.4.2 A Comparison to GNATS

Many of the people who responded have had experience with the GNATS system. Since it is one of the more usable systems available, this section focuses on it.

Primary attributes which GNATS handles well:

- Good incorporation with email
- Solicits relevant information clearly and rigorously
- Low overhead to view and alter bug reports

All of these features are present in my design and are explained in following sections. The solicitation of bug report information in my system is a function of *sendbug* which can be easily altered to include more information.

The following is a list of areas where GNATS needs improvement:

- Faster and simpler queries
- More query functionality
- A linkage between the reports and the code with which they are associated, so that one can see the progress which has been made to fix the bug

- A smaller input questionnaire, with less redundant information and more field consistency, as well as easier customization of that questionnaire
- A better maintenance interface; the current one requires editing rather than being request-based, which seems more robust and scriptable
- Interface to a user call support service
- Better handling of historical archiving and indexing of older bugs

Section 3.9.3 provides an explanation of how my system might accommodate chaining. My system was planned to deal with the listed weaknesses and to provide an easy, command-line maintenance interface.

Because it would depend on the specifics of the other system involved, and the research of user call support systems is not within the scope of this project, I have not addressed the issue of interfacing my system with other systems. However, if such a system could be established with a generic report structure, then one could create another database group, as discussed in section 5.1.2, and interfacing would not be a difficult task.

1.5 The Plan

The scope of this project, if implemented to its completion, is far more than one could do in the time allotted for this thesis. Therefore, I proposed to build a system which has the basic features of a bug tracking system: bug entry, organization and assignment, security, and archiving. I proposed to implement a client-server pair with the above functionality and a design that could support additions which address the concerns stated above. I will have succeeded if I can present a functioning client/server pair that accepts, categorizes, accesses, and archives bug reports, and I can explain how the design will easily accept the further features.

Chapter 2

The Design

This chapter describes the design of the main module for the project, the considerations which were taken for each object, and the choices I made before the actual implementation of the project.

Originally the project was to be written in C++ and it was intended to be only a functional stub of the system exhibiting the ability to eventually include the features described in the introduction.

Figure 2-1 provides a graphical representation of the dependencies of the modules involved in the design.

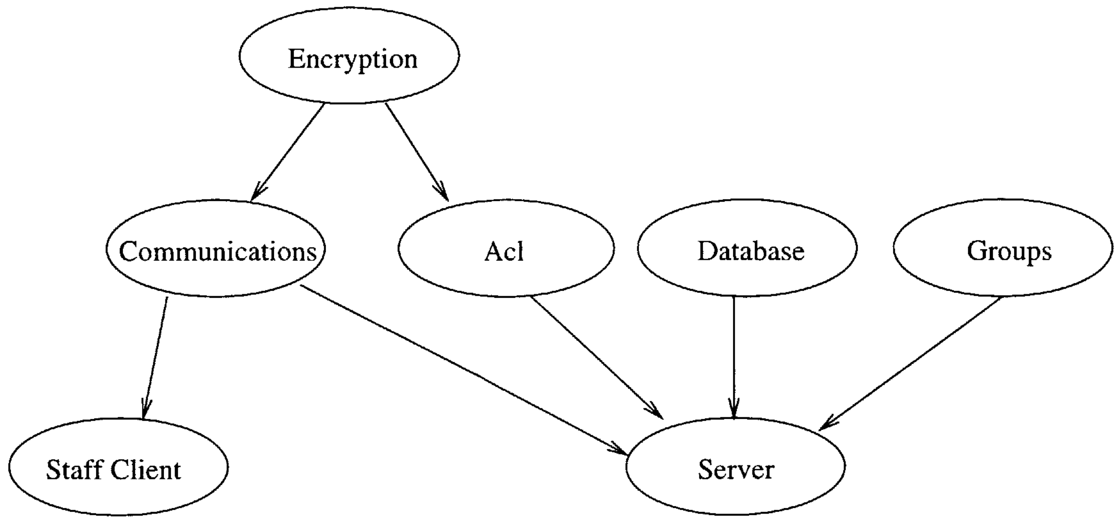

Figure 2-1: Modular Dependency Diagram for the Initial Design

2.1 Generic Modules

This section describes the individual modules that will be used by both the client and the server.

2.1.1 Encryption

Since privacy is a concern, the transmissions between the client and the server need to be encrypted. I chose to use Kerberos[6] (a shared key encryption method) as a convenient tool to do the encryption, but since I do not want to force the production system to use Kerberos, or any single encryption method (since each has different difficulties, limitations, and level of security), this part of the project will have four basic functions: *get_key, encrypt, decrypt* and *authenticate.*

The *get_key* module is called with a server name and acquires a shared encryption key between the client and the server specified. This allows clients to acquire keys for communication with the server, as well as servers acquiring keys for communication with other servers. It is assumed that a server will not wish to use encrypted communication with a client which has not already established a connection.

Once the key has been acquired, the client or server can use *encrypt* (giving it the key, the sender, and the message), to prepare a message for transmission. Once received, the server can use *decrypt*, giving it the sender and the message. Internal to the encryption object, there is a key table that will hold a mapping of all senders to their shared key.

Authenticate will verify that a request made by a client or server is an actual request from that privileged server. This takes a sender name and an authentication string and returns a boolean affirmation of that sender's authenticity.

2.1.2 Communications

Since both servers and clients will want to make information requests to a server, the communications module should be universal. The module should contain all the hand-shaking required to make a request, format it correctly for the server, and to wait for the reply. It should also deal with timeouts, the acceptance of large amounts of data which has been sent in smaller chunks, and any other error checking or security overhead.

The two main modules are *send* and *receive*. *Send* takes a destination and a message, and returns a status. *Receive* takes no arguments and returns a message, which may be an error report from the authentication module signaling failure.

2.2 Server

The server is expected to be run remotely, connected to the clients via the Ethernet, and to service requests from many different clients simultaneously. This means that the design needs to be efficient and have the potential for simultaneous processing. It also needs to be robust, and able to survive power outages, system failures, and other interruptions in service without loss of data. It should be easy to maintain and configure the system, and produce backups.

The server consists of a database and a process which accepts and deals with requests.

2.2.1 The Database

According to *Database Design and Implementation* by L.A. Maciaszek, the data storage system for this server is not a database but an "information retrieval system."[5] For this reason, the design of the database has none of the classical database structures. Since there is little cross-referencing of information, the system can be simple. Any cross-referencing of whole reports can be provided more efficiently with other mechanisms.

The basic functions of the database module are to add a report, get a report at a client request, run a search, and update all time-related attributes of a report. When a report is added, there will be a set of defaults for any field which the submitter of the report did not specify. When a report is requested, the server will process the client request and return the report. When a search is requested, the server will use whatever abilities it has to return the set of reports which satisfies the search. Lastly periodic report updates will happen during times when the server is not being used or on a nightly basis.

2.2.2 Report Groups

To truly understand a bug, one needs as much information as possible. Report grouping allows for related information to be organized for easier viewing, to help the person fixing the bug. Grouping should be as automatic as possible while still preserving sane groups so that time is not wasted creating and maintaining the groups. There should also be different strengths of groups so that, for instance, a direct response to a particular report can be more closely grouped than two reports which merely have a few attributes in common.

Any grouping which is preserved, will be done with a group object which contains references to the reports. The basic functions will be to add a report to a group, remove a report from a group, and show all reports in a group. Reports can be in more than one group, so each report will have a list of groups to which it belongs.

There should also be an automatic grouping process whereby groups are prepared, periodically, for human confirmation. As it gets better, less human intervention will be needed to check its work. This module can be simple, or it can use a very intelligent, complex algorithm.

2.2.3 Access Control List Manipulation

In conjunction with the transmission security, and authentication that the generic security model gives, there is an access control list (acl) system to allow selective reading and writing to the database. Since this is a very useful, but non-essential feature, the implementation could be simple; the acls could be merely lists of authorization names. Also, since other systems for acling (such as MIT's Moira) have already been developed, this feature, as well as other extended features, such as recursive lists[1] (where lists may include both users and names referring to other lists) is available by adopting one of these systems.

In this project, acls could be a list of non-recursive strings which, in conjunction with the security module's authentication, can be looked up to verify that a client has the correct permissions for the function it is requesting. The basic functions of the module are *add, list, remove* and *check.*

Some sort of acl caching should eventually be added so that the acls are not checked for every transaction of a similar nature.

2.2.4 Configuration Files

Since the server configuration files should not, in practice, be changed very often, the configurations can simply be files kept directly on the server machine, in a flat file. The maintainer of the system can alter the files by hand until more flexibility is needed. At that time, an extra maintenance client could be written for more convenience.

The set of configuration files can be changed by a remote client. Reasonable defaults will be provided for unspecified fields. These higher level files can be treated differently in the system, and can control any aspect which should be a highly adjustable component of the server. Examples include identification of fields which are by default in a report, levels of priorities a report can have, how reports are indexed in the database, or any other feature which needs to be a configurable system default.

2.3 Staff Client

The staff client is the client being used by the people who are maintaining the software, or those who need to access the reports and do the investigations of the bugs. Since the staff client will be mostly using the modules generic to the system, there are only

[1]This adds extra complexity since it is necessary to differentiate between users and list names, resolve the list names and not get into infinite resolution loops, as well as other technical problems.

a few modules which need to be explained here. The staff client uses the encryption and communication modules. In an effort to make the staff client as universal as possible, it is independent from any complex user interface. Because of this it only needs a simple way of displaying the information received by the server in either a convenient human or computer readable format.

So as to produce a system that is convenient to use, the design specifies a basic user interface embedded in the staff client. To facilitate this, the client also has *get_command* and *process_command* methods.

The other requirement for the staff client will be to read, create, and alter the personal configuration files local to the client. These files will affect the default information that is included in a request, as well as add and control any added features that the client might have. Since the client will be, in normal usage, restarted more often than the server, it should not share the configuration file modules, even though their basic functionalities will be similar. The overall designs, therefore, make optimizations specific to each application.

2.4 User Client

The user client is used for submission of the original reports and possible viewing of the reports' progress within the system. Currently, the user client has a simple design which brings up a formatted email message template and requests that the user fill in the blanks. Later in the evolution of this project, the user client could be more structured, and allow the user to follow up on the report, be presented with a list of stock answers, or view reports similar to his problem. This would be an easy extension to the requests made by the user client, with modified acling, and can fit conveniently into the existing design described here, but it is not part of the ultimate thesis.

Chapter 3

Implementation

3.1 Introduction

The previous design chapter illustrated the preliminary design and the initial decisions made, as well as describing some of the modifications to the plan. This section explains the actual implementation and the design decisions made during the process. The system described here was implemented in Perl.[4]

For a quick picture of the dependencies of the actual modules, see Figure 3-1.

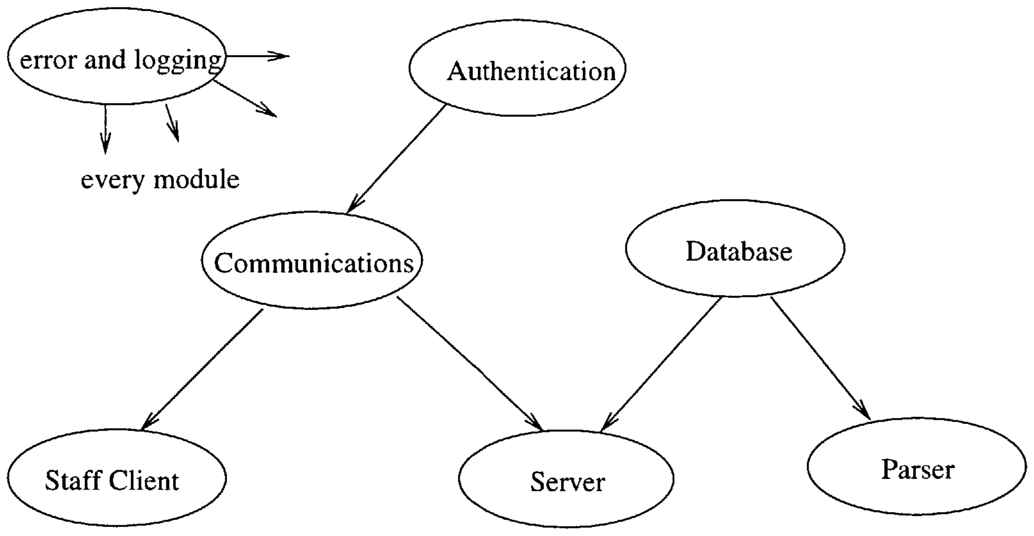

Figure 3-1: Modular Dependency Diagram for the Final Implementation

3.2 Parser

The parser is a simple script which takes incoming reports in the form of email, and tries to canonicalize them for the database. It also adds report specific information, such as a tracking number and default priority.

Initially there is an email message created by a program which prompts the user for information, and sends the email to the proper data logging address. For the purposes of this system, it was configured to work with the existing *sendbug* in the Athena environment.[1]

The server machine's *sendmail* was configured to deliver the emailed report to the script. The script then adds the prerequisite values, and tries its best to get useful information from the report, and assign it to fields.[2] I used the assumption that the report would be delivered in a format which would be useful to the parser (arranged predictably by *sendbug*), but the system is also flexible enough to take non-formatted email and attempt to categorize as much information as possible, for the purposes of classifying it in the database.

The script is configured by a list of lists, implemented as a two dimensional array, which appears at the top of the code in Appendix A.1. Each internal list consists of three parts. The first is a Perl regular expression which is expected to match a line in the email. For example, when *sendbug* sends a subject line such as "Subject: sgi 8.1.15: emacs," the script is configured to pull the platform, release number and application name out of that line and place them into the proper fields. In the configuration section of the code one can express an action which would correctly classify and install those fields. For example, if a line matches "Subject: (.*):(.*)," one can specify that the information found before the colon be put into the field "platform" and the information after the colon be placed in the field "application." Lastly, one can specify what the current default field should be. This default field

[1]All of the testing was done with hand-composed email, since that was the easiest way to mimic *sendbug*.

[2]Fields are, simply, labeled pieces of information. For instance, the piece of information "Calista Tait" would be categorized and placed in the "name" field.

is used if none of the regular expressions in the list of possible expressions matches. The default field starts out as "unknown," and can be changed so that the system can classify multi-line fields correctly. For instance, text describing what actually occurred when the bug was triggered is highly likely to be more than one line. Since one wants to make sure that the entire explanation gets classified, one would match an initial line which would signal the beginning of such a description and then set the default field to "what happened" or other standard name for the field which is to contain all the description. This allows it to be grouped properly without losing embedded line break information.

Because the second part of the configuration is an action, this can be used to discard unwanted information by making the action null. In normal email there are many fields, such as the "Message-Id:" field which are not relevant in most contexts.

Even though all this configuration is in the parser script itself, it will be fairly easy for a maintainer of this system to add any additional expressions and actions, because it is at the top and only requires simple alterations. Most actions do not require writing any code, just writing the field name and the information to be placed in that field.

3.3 Database Interface

For the purpose of this implementation, I used Berkeley DB (a database management program) and its interface to Perl as the database. Most of this implementation was simply applying a standard abstraction over the given database functions, since the flexibility at that level was important. The rest of this section explains the items of interest which went beyond that.

3.3.1 The Database Configuration

The database module dealt with a set of Berkeley DB files, all of which retain the same information, but each of which has a different field as the key. To make things easier, the files are named in a header file. The maintainer can specify a short name which will be used as the actual file name, and the internal name between the servers and the database function calls. A longer "translation name" can be specified which matches the field name used in the internal protocol and format of the report, which can be longer and clearer. This field name will also be used by the client to convey the classification of the information in that field. For example, a Berkeley DB file which is indexed by the field "tracking number" will have a translation name which is exactly that, but internally it may be referred to as "by_no."

Because there are multiple files, the function *create* goes through the list of all Berkeley DB files and creates each one. It then puts the reference returned by the creation into an associative array indexed by the short name. This allows for flexibility in performing functions on all the Berkeley DB files as a group, since to perform an action on all files, one need only cycle through the array, performing the action on each reference contained in it.

3.3.2 Put and Putcommit

This implementation is a wrapper around the provided Berkeley DB functions. The provided functions which were used are *put*, which places a value in the database, *get*, which retrieves a value, *sync*, which forces a write to disk, and *seq*, which would sequentially go through the entire database. Instead of writing a *put* function and a separate *commit* function, I chose to write *put* and *put-commit* functions, where the *put-commit* function does both the *put* and a final *sync*, since I could not think of an application which would need a *commit* that would not also need a *put*. The *close* function takes care of the final write to disk, upon cleanup.

3.3.3 Add

Add takes a bug report and inserts it into all of the existing Berkeley DB files. It uses the field specific to that file to reference it. For each file, it takes the translation name and finds the line which has been denoted by the parsing function. It uses the line which is denoted by that name, as the key for the entry into the Berkeley DB file, and then the remaining data is used for the value. The remainder of the report retains the delimiters on the beginning of the line. For an exact printout of how these values look, see appendix B.2. This is inefficient in several ways. A relevant revision of the system is discussed in section 4.2.2.

3.3.4 Select

This function selects a collection of reports from the database. I designed this function to take advantage of any efficiencies that the underlying database mechanisms would allow. The function accepts a short name specifying which Berkeley DB file to search in, a search pattern, and a range. It has been made as generic as possible so that it could serve multiple uses, such as listing all reports or listing some subset of reports, for example, those which include the word "more."

Because of the way Perl interfaces to Berkeley DB, specifying a range does not help with the efficiency of a lookup. It is still an important feature, however, because once the database gets big, the user will not want to search all requests but some subset of them. Another database system might be better able to use the range function.

3.4 Communications and Authentication

This module takes care of the transmission of requests from the client to the server and the transmission and receiving of the reply from the server to the client. Originally I wanted to do this with TCP/IP[10] sockets, but that proved more difficult than I had originally estimated, so I decided to use Zephyr[3] to serve for testing the validity of the design.

Zephyr was set up to act as the lower level protocol, while ignoring most of the advanced features it provides. I did, however, take advantage of the authentication built into Zephyr. In this implementation, the authentication check would simply

[3]MIT's instant messaging system.

verify that the message had been received with a valid Zephyr authentication flag.

The module is intended to be shared by both the client and the server to ensure consistency, but internal specializations were necessary. The *init* procedure, for instance, had to be different for each, since the server had to acquire server specific tickets,[4] where the client could simply use the user's tickets. This would also be a consideration when using TCP/IP sockets, because the client has to initiate the connection, whereas the server just needs to listen and then do the right thing when it receives a connection. The *send* function also required client and server specialization, since the client is always sending directly to the server, while the server is returning a message to one of many clients which may have contacted it.

The *maintain* function is used for upkeep tasks. In the case of Zephyr, it would serve to renew tickets. It can have other functions when used for other connection styles. In general, most communication methods need some function to verify if the connection is still established, to check if there is something waiting to be received, or to determine if the connection otherwise needs to be maintained at times other than when there is a specific send or receive.

The *receive* function is fairly straightforward. It returns a connection identifier as well as the received message, so that a response may be sent to the correct client. In the case of Zephyr, the connection identifier is just a string containing the username of the sender; in a socket implementation, it would be socket address information. Since the user identifier is used only by the communication module, it can be simply a string with the needed information, which is correctly interpreted by the *send* function.

Lastly there is the *close* function. In this case, it serves more as the cleanup function (which prepares for the process to exit by ensuring that all processes that have not already been taken care of are properly closed or dismissed) than as the close function (which simply closes a current dialog),[5] because with Zephyr it is not

[4]Tickets is the name given to the shared key distributed by Kerberos, which is used by Zephyr to authenticate the sender.

[5]In most cases the server must always continue to listen for new requests. When a request is received, a communications channel needs to be created between the server and the client until the request has been satisfactorily serviced, at which time the server closes the communications channel but keeps listening for new requests. Upon cleanup, the listening channel will also be closed.

necessary to close the connection after the information has been received. It would be more generic to have two functions, *close* and *clean_up*.

3.5 Server

The server is a process which is always running, to receive and service client requests for reading and altering the database.

3.5.1 Dispatch

Dispatch is the function within the server, which takes a request and determines how to service it. This *dispatch* function runs through a set of regular expressions, attempting to match the received request, and then executes a function associated with the matched command. The *dispatch* function itself is fairly robust. To add additional commands, the maintainer adds a regular expression and action pair, and then a function definition, if required, for that action. This makes extending the system easy, though not trivial.

3.6 Client

The client is as plain as possible. It is a human-readable, as well as a computer-readable, tool which communicates with the server. The client, as of this writing, does not have any internal checks for correctness of commands,[6] nor does it have user-configurable manipulation of the output.

The client is text-based. It gives a short startup message and a simple "request:" prompt. It accepts a single line input, sends it to the server and receives the reply. The data sent back from the server is printed to standard output. A lot of the

[6]For the purposes of developing the system, it was easiest to leave this out, since the server was not going to be overloaded by my testing, and I did not have a perfect way to keep the commands available to the client and the actual functionality of the server synched. In retrospect, the dispatch configuration array used in the server could be made available to the client to do a preliminary validity check, while keeping the functionality and client checks in sync.

simplicity, here, is by design, since such pure text can easily be read by any other interface, and then the user interface can be very simply changed. For more details on how the client looks, see appendix B.2.

3.7 Logging

For the sake of simplicity, logged error and status messages from all modules were simply written to a local file. Later, a more robust and common system could be used.

3.8 Protocol

The internal, high level protocol is very simple, but overly verbose in an attempt to be clear, flexible, robust and easy to debug. Some of its inefficiencies can be corrected, however, as mentioned in section 4.2.5. Messages are transmitted in a line-based format where each line is prefaced with the classification of that line. For example, the client sends all requests with "request:" before the request, and all errors are transmitted lines with "error:" at the beginning of the line. This is also used for transmitting reports and their field names. In those cases, the field classification names would preface the line. Any field which is not yet identified, is prefaced with "unknown:" and all other lines are similarly labeled. If multiple reports are being sent, they would be separated by a line matching "–Entry start–" followed by a blank line. This delimiter could also be used to separate summary information as well as full reports. In such a case, selected fields and partial information will be transmitted. This can be indicated with the word "partial" and then the field name. For further examples of how the protocol looks, see appendix B.2.

My intention was to make it very easy to extend the number of fields, since one could just add a new line with a new preface. Some special allowance would be made to prevent new fields from conflicting with keywords, such as "error". This format means that it is easy for fields to hold multi-line, tabulated, and white space formatted lines.

I chose to use plain text, because that made debugging simpler. This requires that more information be transmitted, but at this stage that is not a large concern.

This protocol is overly simple. Further discussion on how to improve it appears in section 4.2.5.

3.9 To Be Finished

3.9.1 Further Database Development

The database is functional, as implemented, but it is missing some key features. One of the simplest is the *change* function, which would have been used to alter reports and make additions to their information in all databases.

I did not implement locking on the databases. If there are multiple servers attempting to alter the same database, the system needs to make sure that the information is consistent. To ensure that one server's changes are completed before another server deals with the database would require locking the databases. Since Berkeley DB, and other database packages, have locking implemented as part of their basic features, this will not be difficult to add.

Also not implemented, but provided for, is the server process which would do nightly maintenance of the database, updating time-dependent fields, compressing older parts of the database, gathering statistics, and sending reminder notices to users. There would also be another database function which would check for newly created databases which did not yet contain older information, and transfer existing information to the new databases.

3.9.2 Acls

Implementing an acling system was beyond the scope of this project, so except for the dummy procedure, no support was put in for it. On every request made to the server, there should be an acl check with the function involved. It could have some number of permissions levels. Each level would have a list of tasks which could be performed at that level, which might include specific functions that could be called or certain reports, classified by attributes, that it would be allowed to affect. An outside system would take the name and the level, and determine if that user had sufficient permission.

There would be an extended command set for the client which would add and remove people from acls, and create and change acl lists. This could then interface to an established acling system.

3.9.3 Chaining

One of the more important features of this system was going to be the ability to chain bugs, with a focus on eventually having an automatic system (probably the nightly maintenance program) to chain the bugs. Most likely I would have implemented this by making another database where bugs which were associated with each other were in a list indexed by the report which was currently being look at. All reports in a group would have a list, indexed by their number, of the rest of the bugs in the group. This creates redundancy, since every group will be listed as many times as the number of bugs in that group. However, since it would be just the tracking number, and most reports would only be associated with a small collection of other bugs, I believe that this redundancy would not cause a problem. It would, however, solve a problem evident in the bug tracking program *netprob*.[7]

3.9.4 Other

The last two features which are not yet included, are configurable fields and reasonable tracking numbers. Some of the configurable field functionality was mentioned in the discussion of the protocol. With that established, the only features which need to be

[7]The *netprob* problem is as follows: Given three reports A, B and C. The user chains B to A and then chains C to B. With netprob you could only see reports which are chained to the head report. So you could see A and B or B and C but not all three, even though the intention is to relate them all.

implemented, are support in the server, and the database *change* function. Lastly, the tracking number is currently the Unix time at which the report arrived. This was quick and functional, however this tracking number should be replaced with something which is easy to remember by a human and is guaranteed to be unique. The suggested replacement would be the current year followed by four letters, a decimal point, and a short number. The letters alone will account for 52 reports an hour. If there are more than that, the short number can be used.

Chapter 4

Evaluation

4.1 What Worked Well

Much of what I implemented worked well, but has already been sufficiently explained. This section will highlight those areas which were not explained in the preceding chapters.

4.1.1 Extensibility and Maintainability

Because of how the code was implemented, it is very easy to add additional functionality, to configure it, and to maintain it. The interfaces for the modules which were designed to be swapped out (i.e. replacing Zephyr with something more specialized, the database with something which deals better with large amounts of data, and the security with stronger protection) were not dependent upon assumptions about the specifics of the modules actually used, as I have discussed. Keeping the parts of the system which needed to be configured separate, and easily extensible, made adding new features after the system was working, very easy and in most cases required little debugging.

4.1.2 Portability

Writing this project in Perl makes the system portable to any Unix system with Zephyr available. Though the system was not tested under MS Windows, Perl is available for Windows, so with a bit of conversion of the Zephyr interface, it can be expected to work effectively.

Since I would like to see this system rewritten in another language, I will point out the inherent portability of the design. The client/server communication is simple, and since the client itself is simple, it will be easy to make clients which can connect and accept the information from the server on a variety of platforms without much complication. The largest problem will be to ensure that the communications module could be written on each platform.

4.2 What Did Not Work Well

4.2.1 Parser

Dissecting the mail message line by line with only the default field as a state keeper was not flexible enough to properly deal with mail messages. Fields which might have multiple lines, and also might precede unmatched lines, could not be accommodated. The addition of a flexible flagging system where one could indicate that a state exists, but should only exist for a limited amount of time, or other, more complex indicators would help this problem. Additionally having the default field revert to "unknown" or some other default, after some number of lines might also help this problem. In general though, the parser did work effectively.

4.2.2 Database

Most of the problems with the database were efficiency issues rather than actual non-functionality.

The first problem is that the *select* function assumed that the keys of the database could be given in a sorted order. Sorting is a costly operation, so instead of sorting each time the function is called there are two solutions. Many databases can be kept in a sorted fashion,[1] or a separate, sorted list of keys can be kept, then when *select* cycles through the keys, it uses that list rather than something provided from the database. The latter might also be in a data structure which ensures that the ranges, which *select* accepts as arguments, can be used optimally to avoid accessing reports which do not need to be accessed. The former would reduce the overhead and space of keeping the list, if the database allows it, but may not be the optimal form for the database for other functions.

The second problem is the method for storing the multiple Berkeley DB files, each with different indexing. This is very wasteful of space. A relational database would eliminate this problem easily. Alternatively, if that is not the option which a maintainer wishes to use, the current database system could simply use tracking numbers instead of keeping entire reports in the other files. This would only slow down access marginally and would reduce space requirements greatly.

Another problem with *select* is that it does not allow specification of what fields to search. I would add another argument, which is a list of fields to search. It is important to be able to specify multiple fields, since I wanted the search to be sufficiently flexible that there would be enough information for an automatic chaining program to be installed. It should also have some facility for doing "and" or "or" specifications on which fields to use.

4.2.3 Perl

Perl is a wonderful language for regular expressions and scripting. It is also good for getting a basic system working quickly. However, because I kept most of the code in modules, the clients and server scripts were hard to move from the original directory because of the code's dependency on its location. This system, if it were to be rewritten, finished, and put into production, needs to be in a compiled language, or at least an efficient language where it is easy to move parts around without large

[1]Berkeley DB, for instance, has an option to store all the keys in a binary tree

dependence on other files. Perl can be compiled, but I feel that C (and C++) and C compilers are more common and consistent over platforms and more people have stronger backgrounds in C-like languages. A C-like language would probably be better than compiled Perl for efficiency and maintainability.

4.2.4 Communications

For efficiency and to keep the communication protocol consistent between the client and the server, the communications module was shared by both the client and the server. This, however, caused other problems. The *send* and *receive* functions both had to be specialized, as described earlier. This made the functions hard to maintain. For instance, there was a mildly annoying situation when the server and the client would not run on the same machine. This is not a large problem, but the simple fix it required could not be implemented without uncoupling the client and server functions entirely. This would be even more of a problem using TCP/IP sockets, because the initialization is even more specialized. These problems are significant enough that in a redesign, the client and the server communications modules should be separate.

4.2.5 Protocol

Efficiency

The protocol used was designed to be easy to debug. There are many places where it could be optimized for transmission speed, rather than clarity. The easiest way to improve speed is to use a unique numerical sequence for each field. Because the system still needs to be flexible, yet systematic, it would still have a delimiter byte (for example, the colon is the current case) but the text (for instance, "subject") can be replaced by a single number. I would constrain the numbers to be an integer number of bytes long, so that it would be easy to detect the delimiter byte.

Robustness

The protocol's flexibility causes a small problem. Part of a good protocol is robustness. My protocol, though it could robustly identify a line, did not enforce any canonicalization of the fields. If the system were to be actually used, users could create a field which was similar to, or exactly the same as, another field. This would cause information to be cluttered and not well organized because of the flexibility of the protocol to gleefully accept new fields, without consideration for repetition.

In addition, robustness could be improved by including some sort of checksum, and line and text lengths. As well as making sure that that a report has not been cut off, these devices could be used by a client to predict how best to format the output. For instance, if a GUI client is told that a field is bigger than the allocated space, it can adjust the space, given the size information, or create a paging mechanism to view all the information. Getting the contents of a field could be done more generically using the line numbers as an indication of when to stop reading information.

Chapter 5

The Future

This section describes the features which this system should have if it were to be used or further developed. These are attributes which are non-essential but are still important. These are also the attributes for which I have reasonable (though not necessarily perfect) suggestions, and ones which would require much more research, development and experimentation to find the optimal solution for the situation.

5.1 Replacing the Current Modules

Since the design had a secondary focus of allowing the user to replace modules with customized implementations, it is not imperative to discuss replacement of the makeshift pieces I chose. I will address this issue, however, with some brief suggestions, because I feel that the current system has acquired too many limitations.

5.1.1 Replacing Zephyr

Zephyr would best be replaced with some specialized code written with this specific purpose in mind. It should be an efficient, low level protocol with provisions for checksums and optional encryption and security. It should also be written so that portability is preserved.

Since in this application, Zephyr also takes care of authentication, it could be replaced with Kerberos methods, as well as using Kerberos for encryption or other security issues.

5.1.2 Database Replacement

One of the possibilities for the database replacement would be a relational database. This will take care of the wasted space problem mentioned earlier which is associated with having several databases which are indexed by different fields. And, as with any of the other replacements, the user can choose from the variety of finished systems to get the options and optimality that they want. If nothing else, however, a reliable atomic commit system should be deployed, so that the database will be consistent and fairly up-to-date after recovering from system interruption due to power outage or fatal machine error. These protocols exist and are a simple and important step toward reliability.[9]

To address the issue of having multiple, but distinct databases, database segregation can be implemented by adding another level of grouping. There is already a field hierarchy in the reports, reports are grouped in the set of databases, and in turn those sets of databases can also be grouped. The *create* function can be called on each group of databases, making them separate, but still accessible by the same set of commands. Another specifier can trivially be added to tell the system which database group is being accessed. Transfering reports between the two groups would require the implementation of another simple function.

5.2 Concurrent Servers

If this system is to be further developed, the server should be enhanced so it can handle parallel processes. Currently, the database accesses are the main concern when considering concurrent processes. They are time-consuming and are usually required when responding to a request. Since there are multiple databases, separate locks could be set on each one, and then two requests, if they concern different databases, could be serviced at the same time. Also, if writing to the database became more efficient, it would then not require long locking of the database. Reads need not hold the same lock as writes to further aid parallelism. Lastly, the communication module, when it gains information from processing a Zephyr, needs to keep the information consistent by parsing a whole Zephyr message without interference from another process.

5.3 Reports and Statistics

One important issue that I have not already addressed, is how to implement reports, statistics, and nightly reminders in the system. It is often important to know how big the database is, what the average lifetime of a bug is, what the average lifetime of the bugs which have not been serviced is, who is doing the most closing of reports, and many other types of information which can be used to evaluate the effectiveness of the bug fixing process and find better ways to service the customer.

Some of these statistics can easily come directly from the database as it stands now. A nightly process, for instance, could count the total number of reports which meet a set of requirements, such as who is working on the report, the number of open reports, the number received today, etc. Other additions could be made, such as "date entered" and "date closed" fields or simply a "time open" field to keep information on the time processed.

Finally, the nightly script, while it is getting this information, can compile nightly reminders to the relevant people, including managers and customers who are concerned about the progress of the reports.

5.4 Scalability

Scalability is, in this case, how well the system performs when both the bug database and the number of users get large. This system's ability to do this will depend on the nature of the database replacement, since the reading of the database is the slowest part of the system. The easiest, most general way to make things more scalable, would be to replicate the database in several places and then have a set of servers, each of which can accept client connections based on load balancing.

Options should also be considered for how to make the large database manageable; for instance, the clients should not, by default, do entire database reads. Older reports should be archived and classified by topic and relevance, so that if someone needs that information it is not lost in the large number of unorganized old reports. Even with search procedures, it is hard to make sense of so much information if it is badly organized.

Chapter 6

Conclusion

In this thesis I have explored the difficulties of making a good bug tracking system, I have taken those common problems and have designed and partially implemented a system which begins to satisfy the needs and concerns for such a system, and I have shown where efforts still need to be made, as well as suggested solutions for the problems which are not yet solved. This design, implementation, and evaluation demonstrates that it can be a relatively direct procedure to build a bug tracking package which is straightforward, adaptable, and extensible, while being easy to use and versatile. The method proposed here is essentially sound, and incorporates a number of important elements which should be included in any good bug tracking system. Considerable further development is required before the proposed method would approach the ideal. However, if the designer of the next system is careful to retain the underlying attributes of simplicity, flexibility, and maintainability in the design, as has been highlighted in this project, then the system can grow with, and adapt to, the many needs of the people who will be using it.

Appendix A

Code

A.1 The Parser

```
#!/afs/athena/contrib/perl5/bin/perl
use lib '/afs/athena.mit.edu/user/c/a/cat/project/thesis/newcode/server';

use db;

# this is the script that parses incoming messages, and places those
# reports into the databases.

# It is assumed that the information is coming in on stdin and is in a
# nice, happy, text format.

# I think that using an multi-dimensional array for reg exps/function calls, is
# better than if statements, so:

#               reg exp                line to add         current field
@pattern_do = (["Subject:  (.*):(.*)", "\"platform:  \$1\napplication:  \$2\n\"", "subject"],
               ["Subject:  (.*)", "\"subject:  \$1\n\"", "subject"],
               ["System name:\w*(\W*)", "\"machname:  \$1\n\"", "machname"],
               ["From:  (.*)", "\"sender:  \$1\n\"", "sender"],
               ["Date:  (.*)", "\"date:  \$1\n\"", "date"],
               ["Message-Id:  ", "", "unknown"],
               ["To:  ", "", "unknown"],
               ["Received:  ", "", "unknown"]
               );

 db::init();

$entry = '';

# add the tracking number
$time = time;
$entry = "tracking number:  $time\n";
```

```
&error::log("sbtp_parser:  Entering $time");

$c_field = "unknown"; #current feild
while($line = <STDIN>) {
   $usedf = 0;
   foreach $i (0 .. $#pattern_do) {
         $tempexp = $pattern_do[$i][0];
         if ($line =~ $tempexp) {
            $usedf = 1;
            $entry = $entry . eval $pattern_do[$i][1];
            $c_feild = $chatarray[$i][2];
            last;
         }
   }
   if (!$usedf) {
         $entry = $entry . "$c_field:  $line";
   }
}

&db::add($entry);
&db::cleanup();

exit(0);
```

A.2 The Communications Module

```
# Welcome to the perl module comm

# this will take care of all your communication needs, including
# authentication, encryption and general transmission of information.

# this is the lower level of the protocol and will have an upper level
# which takes care of field delimiters and various other infomational
# protocol aspects.

# I have combined the server and the client communications here because
# they will only differ in how they initially get a connection.

# For example, in using zephyr, they will not differ at all, since each
# new write will require a new zwrite process and the client and the
# server will both have to have a zwgc running.  The client, however,
# will have to identify itself to the server.

# in a lower level implementation with raw TCP/IP sockets, however, the
# difference will have to be much different, since the client will
# have to establish a socket with the server and the server will have
# to accept that connection.

# when converting to c++, one might just replace zwgc and zwrite with
# a lower level package only seen by the shared code.
```

```
package comm;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(clientinit serverinit read write update);

use header;
use security;
use acl;
use error;

# declaring package variables.
$readhandle;
$serverflag = FALSE;

sub clientinit {
   $mdir = $header::home;
   # use the user's tickets.

   # start zwgc
   $ENV{WGFILE} = "/tmp/wg.sbtp";
   open(ZWGC,
         "/usr/athena/bin/zwgc -nofork -ttymode -f" .
         " $mdir/.zwgc.desc -subfile $mdir/.zephyr.subs|");
}

sub serverinit {
   #get server tickets
   &security::init("sbtpserv\@ZONE.MIT.EDU");
   # for zephyr, you just wnat to start zwgc as well.
   $serverflag = TRUE;
   &clientinit();
}

sub maintain {

   # TCP/IP needs to check and
   # send the things that are in the buffer.
   # this function will also renew server tickets.

   if (serverflag) {
         #check tickets (or ticket timer?)
         #renew if expired
   }
}

sub csend {
   local($message) = @_;
   &send("sbtpserv\@ZONE.MIT.EDU", $message)
}

sub send {
   # arguments: string message
   local($to, $message) = @_;
    open(ZOUT, "|zwrite -c sbtp $to > /dev/null")
```

```
        or &error::log("Send Open failed:$!\n");

    print(ZOUT $message);
    close(ZOUT) or &error::log("Send Close failed, perhaps server is down");
}

sub receive {
    # arguments:
    # returns: string, connection identifier
    # on error returns: permissions denied

# So I have to do something here that remember who we are talking
# to.  The problem with this is that there is only one return value.
# I could send a reference, as I would in C, but I get the feeling you
# don't want to do that in perl...  I could keep it around in some
# magic array, or return an array.... neither sounds all that good.  I
# need to think about this, which is why I'm writting this here,
# trying to sort things out.

# I could make the requirement that I'm only accepting one connection
# at a time, therefor the current connection could be kept internal
# to the procedure.  This would be reasonable for the server, but
# limits things.  Maybe some queueing system...

# I really want something that will allow for servicing of requests
# while waiting on a request that might take a long time...  but maybe
# this is not reasonable.


    # since perl treats zwgc as a file handle, it takes care
    # of all the buffering that needs to happen for TCP/IP.  This is
    # where we would just pull out what we could from the buffer.  }
    $class = "";
    $instance = "";
    $body = "";
    $sender = "";
    $sig = "";
    $time = time;

    $_ = <ZWGC>;
    while(!/done/) {
        if (/sender: (.*)/) {
            $sender = $sender . $1;
        }
        if (/body: (.*)/) {
            $body = $body . $1 . "\n";
        }
            if (/auth: (.*)/) {
            $auth = $auth . $1;
        }
             if (/class: (.*)/) {
            $class = $class . $1;
        }
        $_ = <ZWGC>;
```

```
  }
  if (($sender =~ /ZWGC/) || ($class ne "sbtp")) {
        # this is to get rid of that annoying first zephyr or any
        # personals
        &receive();
  }
  if (&security::auth_check($sender, $auth)) {

           return(($body, $sender));

  }
  else {
        return(("comm:  permissions denied", $sender));
  }
}

sub close {
  close(ZWGC);
}
```

A.3 The Support Files for Zephyr

A.3.1 .zwgc.desc

```
if (downcase($opcode) == "ping") then
        exit
endif

case downcase($class)
default
        fields signature body
        if (downcase($recipient) == downcase($user)) then
                print "personal\n"
        endif
        while ($opcode != "") do
                print "opcode:" lbreak($opcode, "\n")
                print "\n"
                set dummy = lany($opcode, "\n")
        endwhile
        while ($class != "") do
                print "class:" lbreak($class, "\n")
                print "\n"
                set dummy = lany($class, "\n")
        endwhile
        while ($instance != "") do
                print "instance:" lbreak($instance, "\n")
                print "\n"
                set dummy = lany($instance, "\n")
        endwhile
        while ($sender != "") do
                print "sender:" lbreak($sender, "\n")
                print "\n"
        set dummy = lany($sender, "\n")
    endwhile
    while ($time != "") do
        print "time:" lbreak($time, "\n")
        print "\n"
        set dummy = lany($time, "\n")
    endwhile
    while ($auth != "") do
        print "auth:" lbreak($time, "\n")
        print "\n"
        set dummy = lany($auth, "\n")
    endwhile
    while ($date != "") do
        print "date:" lbreak($date, "\n")
        print "\n"
        set dummy = lany($date, "\n")
    endwhile
    while ($fromhost != "") do
        print "fromhost:" lbreak($fromhost, "\n")
        print "\n"
        set dummy = lany($fromhost, "\n")
    endwhile
    while ($signature != "") do
        print "signature:" lbreak($signature, "\n")
        print "\n"
        set dummy = lany($signature, "\n")
    endwhile
    while ($body != "") do
        print "body:" lbreak($body, "\n")
        print "\n"
        set dummy = lany($body, "\n")
    endwhile
    print "done\n"
    put "stdout"
    exit

endcase
```

A.3.2 .zephyr.subs

```
sbtp,*,%me%
!message,personal,%me%
```

A.4 Server

A.4.1 srvh.pm

```
# This file contains the things that the three servers and their sub
# modules need to know.

package srvh;
```

```
use lib '/afs/athena/user/c/a/cat/project/thesis/newcode/share';

# this is a list of all the databases.  For a system, one should add
# any field which is used to search on regularly.

@databases = ("by_no", "by_app");

# this is for field translations that might occur between the
# sendbug type application and the server.  This will allow for a bit
# of flexibility in the application, without having to redo the
# databases.

# the format is field name, database name

%field_tr = (
            'by_no', 'tracking number',
            'by_app', 'application name'
            );
```

A.4.2 db.pm

```
package db;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(add init archive delete);



use lib '/u2/DB_File-1.60/blib/lib';
use lib '/u2/DB_File-1.60/blib/arch';
use lib '/u2/DB_File-1.60';
use DB_File;

use lib '/afs/athena/user/c/a/cat/project/thesis/server';

use srvh;
use error;

$dbdir = "/u2/sbtp/";

%dbarray = {};


# to add:
#       if a new database is introduced (zero length) but other
#       database has info, have it update
#
#       locks

sub create{
   local($name) = @_;
   $temp = dbmopen %{$name}, $dbdir . $name . "_db", 0600;
   if (!$temp) {
        &error::log("db:  create:  error opening dbfile $dbdir$name" . "_db");
        exit(-1);
    }
    $dbarray{$name} = $temp;
}

sub close{
    local($name) = @_;
    dbmclose(%{$name});
}

sub put{
    local($name, $key, $value) = @_;
    $db = $dbarray{$name};
    if ($db) {
        $status = $db->put($key, $value);
        &error::log("DEBUG: calling put with $key and $value");
    }
    else {
        &error::log("db:  put:  no such database $name");
    }
}

sub putcommit {
    local($name, $key, $value) = @_;
    $db = $dbarray{$name};
    &put($name, $key, $value);
    if ($db) {
        $status = $db->sync();
    }
    else {
        &error::log("db:  putcommit:  no such database $name");
    }
}

sub get {
    # returns: $value
    local($name, $key) = @_;
    $db = $dbarray{$name};
    $status = $db->get($key, $value);
    return($value);
}

sub add {
# This takes an entire bug report in "system internal std format" and
    #adds it to all of the databases

    # $entry is a string with the data


    local($entry) = @_;

    # I'm going to make this inefficient for the sake of complexity of code.
    # I'm sure that this part could be rewritten to only take one pass at the file.
    @lines = split(/\n/, $entry);
    while (($name, $db) = each %dbarray) {
        # for each database
        # find it's offical feild name
        $fieldexp = "^" . $srvh::field_tr{$name} . ":   (.*)";
        $key = '';
        $value = '';
        foreach $line (@lines) {
          if ($line =~ $fieldexp) {
                # incase the key is multiline
                $key = $key . $1;
          }
          else {
                $value = $value . $line . "\n";
          }
        }
        putcommit($name, $key, $value);
    }
}

sub select {
    local($name, $reg, $start, $stop) = @_;

    # so this doesn't make much sense in an unsorted db, but we could
    # switch to B_tree and all would be grand.

    # to be finished: through all of the keys of a db and return the
    # ones which match the reg exps.

    $db = $dbarray{$name};
    $out = '';
    $status = 1;

    # seq reportably starts at the beginning and goes from there, so we
    # need to cycle until we hit the start place.
    # yes this is inefficient.

    &error::debug("select:  starting seq.    reg:  $reg");

    $status = $db->seq($key, $value, R_FIRST);
    &error::debug("db:  select:  init status:  $status");
    &error::debug("db:  select:  key:  $key start:  $start stop:  $stop");
    while (($key ne $start) && ($status != 1)) {
        &error::debug("select:  seq until start\n\tkey:  $key\n\tstatus:  $status".
                    "\n\tstart:  $start");
        $status = $db->seq($key, $value, R_NEXT);
    }

    while (($key ne $stop) && ($status != 1)) {
        if ($value =~ $reg) {
          $out = "$out--Entry start--\n\n$value";
        }
        $status = $db->seq($start, $value, R_NEXT);
        &error::debug("select:  seq until end");
    }
    return($out);

}

sub first {
    local($name) = @_;
    $db = $dbarray{$name};

    $start = '';
    $db->seq($start, $value, R_FIRST);

    return($start);
}

sub last {
   local($name) = @_;
    $db = $dbarray{$name};
    $db->seq($start, $value, R_LAST);
    return($start);
}

sub init {
    # opening it all.
    foreach $dbname (@srvh::databases) {
        &create($dbname);
    }
}

sub cleanup {
    foreach $dbname (@srvh::databases) {
        &close($dbname);
    }
}

sub change {
    # takes old, new

    # this will be for changing fields of an existing entry.  In some
    # DB's this will be the same as rewriting the entry, but in others
    # it might be a more elegant change.

    # makes changes in ALL databases

}

sub update {

# this procedure should go through all of the old entries and reinsert
# them into to any new database separations.  This should only be
# called if people have changed which databases are relevent, but want
# to make old database information fit into the new database
# structure.

}
# pm files need to return a true value... go figure.
1
```

A.4.3 server.pl

```
#!/afs/athena/contrib/perl5/bin/perl
use db;
use comm;
use error;

&comm::serverinit();
&db::init();


@dispatch_a = (["request: find (.*) in (.*)", "&find(\$1, \$2)"],
            ["request: list all", "&list_all()"],
            ["request: list_by (.*)", "&list_by(\$1)"],
            ["request: get ([^ ]*) ([^ ]*)", "&db::get(\$1, \$2)"]
            );

&error::log("Server started");

while (1) {

  # this should eventually get some signal handling so it can clean
  # up on exit, but for right now this will work fine.

  local($string, $con_id) = &comm::receive();
  &error::debug("server: mess: $string \nid: $con_id");
  if ($string ne "permissions denied") {
      # I should make the above test more robust
      $out_s = &dispatch($string);
      &comm::send($con_id, $out_s);
  }
  else {
      &comm::send($con_id, "error: permission denied");
  }
}

sub dispatch {
  local($line) = @_;

  $usedf = 0;
  foreach $i (0 .. $#dispatch_a) {
      $tempexp = $dispatch_a[$i][0];
      &error::debug("server: dispatch: '$tempexp' matching '$line'");
      if ($line =~ $tempexp) {
         $usedf = 1;
         &error::debug("sever: Dispatch: evaluating $dispatch_a[$i][1]");
         $response = eval $dispatch_a[$i][1];
```

```
            last;
        }
    }
    if (!$usedf) {
        return("error:  Unknown request\n");
    }
    else {
        return($response);
    }


}

sub list_all {

    # list all, which in general will probably not be used
    # often, will list things by tracking number.
    &error::debug("listing all");
    return(&list_by("by_no"));

}

sub list_by {
    local($cat) = @_;

    # the client will be expected to have translated things into the
    # cannonical category names
    &error::debug("list_by:  calling select");
    return(&db::select($cat, ".*", &db::first($cat), &db::last($cat)));
}

sub find {
    local($reg, $cat) = @_;
    return(&db::select($cat, $reg, &db::first($cat), &db::last($cat)));
}
```

A.5 Client

A.5.1 ch.pm – the client header file

```
package ch;

use lib '/afs/athena/user/c/a/cat/project/thesis/newcode/share';

# grr... perl want's true, I'll give it true
1
```

A.5.2 client.pl

```
#!/afs/athena/contrib/perl5/bin/perl
use ch;    # client header
use comm;

#initialize the client/server connection
&comm::clientinit();

$quit = 0;

print "stpb client started\n";
while(!$quit) {

# until we're asked to quit will take the requests and feed them to the
# server.

#for now, we'll let the server deal with checking valid requests.  For
# the interest of load, the client should ultimately do some checking.

    print "request:  ";

    $request = <STDIN>;
    if ($request =~ /^quit/) {
            $quit = 1;
    }
    else {
            &comm::csend("request:  " . $request);
            local($string, $con_id) = &comm::receive();
            print($string . "\n");
    }
}

&comm::close();
```

A.6 The Shared Header

```
# This is a file which contains all the configuration information for
# the shared packages.

package header;

$home = "/afs/athena/user/c/a/cat/project/thesis/newcode";

# grrr...
1
```

A.7 The Error Module

```
package error;

require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(log debug);

require "ctime.pl";
require "timelocal.pl";

$debug_f = 1;

sub log {
   # this will syslog when the system is a bit more up and running
   local($message) = @_;

   $time_s = &ctime(time);
   $time_s =~ /([\w]+) +([\w]+) +([\d]+) +([\d]+:[\d]+:[\d]+)/;
   open(LOG, ">>/var/adm/sbtp.log");
   print(LOG "$2 $3 $4:  $message\n");
   close(LOG); }

sub debug {
   local($message) = @_;
   if ($debug_f) {
         print "DEBUG: $message\n";
   }
}

# pm files need to return a true value... go figure.
1
```

A.8 The Acl and Security modules

A.8.1 acl.pm

```
package acl;

sub acl_check {
   local($user, $permission) = @_;
   #just a stub for now.
   return(1);
}

1
```

A.8.2 security.pm

```
# this is the authentication checking mechanism and encyrption.
```

```
package security;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(init auth_check);


# for kerb, we'll assume everything we need is in a srvtab for now.
$srvtab = "/var/cat/sbtp/sbtp.keyfile";

sub auth_check {
   # arguments: string sender, string authenticator
   # returns: boolean
   # tests the authenticator and returns true if the send is authentic
   return(1);
}

sub init {
   local($id) = @_;
   system("kinit -k -t $srvtab $id");
}

# grrr..
1
```

Appendix B

Examples

B.1 A Typical Athena Sendbug Script Output

```
To: bugs@MIT.EDU
Subject: sgi 8.2.15: netscape
-------
System name: oliver.mit.edu
Type and version: IP32 8.2.15 (with mkserv)
Display type: CRM

What were you trying to do?
Quit netscape.

What's wrong:
Instead of exiting gracefully, it reported ``bus error'' and
dumped core.

What should have happened:
It should have exited gracefully and not given me a large core
file.

Please describe any relevant documentation references:
None that I can think of.
```

B.2 An Example Transcript of Using the Client

This is only a few of the commands, but it gives an idea of both how the client requests and receives the data, and how the internal protocol looks, since the client simply prints out the text of the internal protocol. The "request:" is printed by the client. The text after it is the user-entered request, and the output after that is the response from the server.

See section 3.8 for an explanation of some of the fields.

```
request: finger more in by_no
error: Unknown request

request: find more in by_no
--Entry start--

unknown: From cat@mit.edu  Sun Jan 10 21:36:56 1999
date: Sun, 10 Jan 1999 21:36:55 -0500 (EST)
sender: Calista E Tait <cat@mit.edu>
subject: This is some MORE test data.
unknown: content-length: 31
unknown:
unknown:
unknown: This is some more test data.

request: list all
--Entry start--

unknown: From cat@MIT.EDU  Sun Jan 24 15:30:33 1999
unknown:        id AA11790; Sun, 24 Jan 99 15:30:48 EST
date: Sun, 24 Jan 1999 15:30:32 -0500
sender: Calista E Tait <cat@MIT.EDU>
platform: sgi 8.2.15
application:  netscape
unknown: content-length: 425
unknown:
unknown:
machname:
unknown: Type and version:      IP32 8.2.15 (with mkserv)
unknown: Display type:          CRM
unknown:
unknown: What were you trying to do?
unknown:        [Please replace this line with your information.]
unknown:
unknown: What's wrong:
```

```
unknown:        [Please replace this line with your information.]
unknown:
unknown: What should have happened:
unknown:        [Please replace this line with your information.]
unknown:
unknown: Please describe any relevant documentation references:
unknown:        [Please replace this line with your information.]
--Entry start--

unknown: From cat@mit.edu  Sun Jan 10 21:36:56 1999
date: Sun, 10 Jan 1999 21:36:55 -0500 (EST)
sender: Calista E Tait <cat@mit.edu>
subject: This is some MORE test data.
unknown: content-length: 31
unknown:
unknown:
unknown: This is some more test data.
--Entry start--

unknown: From cat@mit.edu  Sun Jan 10 21:36:39 1999
date: Sun, 10 Jan 1999 21:36:38 -0500 (EST)
sender: Calista E Tait <cat@mit.edu>
subject: This is some test data.
unknown: content-length: 26
unknown:
unknown:
unknown: This is some test data.

request:
```

Bibliography

[1] Angelo Renato Bobak. *Data Modeling and Design for Today's Architectures.* Artech House, Boston, Massachusetts, 1997.

[2] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[3] Reinhard Helfrich. Software error administration using the gnats system. Technical Report 10355, European System and Software, Paris, France, November 1995.

[4] Randal L. Schwartz Larry Wall, Tom Christansen. *Programming Perl.* O'Reilly, Cambridge, Massachusetts, second edition, 1996.

[5] L. A. Maciaszek. *Database Design and Implementation.* Prentice Hall, New York, New York, 1990.

[6] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[7] G. M. Nijssen and T. A. Halpin. *Conceptual Schema and Relational Database Design: A Fact Oriented Approach.* Prentice Hall, New York, New York, 1989.

[8] A. Udaya Shankar. Modular design principles for protocols with an application to the transport layer. Technical Report 2510.1, University of Maryland, College Park, Maryland, July 1990.

[9] James W Stamas and Flaviu Oustian. A low-cost atomic commit protocol. Technical Report 7185, IBM Research Devision, Yorktown Heights, New York and San Jose, December 1989.

[10] W. Richard Stevens. *Networking APIs: Sockets and XTI*, volume 1 of *Unix Network Programming*. Prentice Hall, Upper Saddle River, New Jersey, second edition, 1998.

[11] Toby J. Teorey. *Database Modeling and Design: The Entity-Relationship Approach.* Morgan Kaufman Publishers, Inc., Palo Alto, CA, 1990.

Rc 11-95