MQP-CEW-0601

CISCO E-MAIL FILTER AND NOTIFICATION SYSTEMS

A Major Qualifying Project Report:

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

_____

Jeremy N. Lounder

_____

Cody B. Rank

_____

Brian G. Weber

Date: May 01, 2006

Approved

_____

Professor Craig E. Wills, Major Advisor

ABSTRACT

A class of software development tools at Cisco Systems currently uses an inefficient e-mail notification system that floods users with information that may or may not pertain to them. This project designed and evaluated a new mail filter system to give users more control over what, when and how they receive e-mail messages. The filter is implemented in Ruby on Rails to receive information from Cisco's tools and then send e-mail messages that have been filtered by user-chosen delivery options.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

In 2003, Cisco Systems, Inc. built a campus in Boxborough, Massachusetts. The intent of this campus was to create a central location which would save workers time when working with other Cisco Departments in the Northeast. In order to boost this cross-department productivity, a system was implemented which broadcasts important information about current tasks to other departments. Current employees of Cisco receive many notifications, via e-mail, about the status of software currently under development. These notifications can be about new software for internal use by Cisco, new software for release, or a updates to existing software. Some of these notifications are important, as they require actions by certain individuals to progress, whereas others are not as important and are more informative than critical.

Bill Lapp and Paul Russell, employees at the Cisco campus in Boxborough, along with Professor Craig Wills, at Worcester Polytechnic Institute, discussed a problem with this system that they felt would be appropriate to be addressed by students at Worcester Polytechnic Institute.

The problem we have been tasked to solve is an important one. In the world of business, time is money and consequently wasted time is wasted money. Any measures that can be taken to save an employee even minutes a day become worthwhile when viewed over the course of a month and hundreds of employees. It is important for users to be able to easily find action items so that they may handle them in a timely fashion and ensure that these items do not get lost in a sea of informative notifications. We have been asked to provide a more efficient way of getting the critical task information out to the people who need it and to give Cisco employees a means to reduce the number of e-mail notifications they receive.

1.1: PREQUALIFYING PROJECT

During the months of January and February, 2006, we were in contact with Professor Wills and Bill Lapp to coordinate details of the project. We had an in person meeting with Bill and Paul Russell at the Cisco Campus in Boxborough, and then we continued communication via e-mail for the rest of the time.

During our meeting at Cisco, we were presented with a general summary of the problem the Cisco employees were facing. They were receiving a flood of automated e-mails from various Cisco applications on a daily basis. Some employees were receiving a hundred or more e-mails per day. Paul was also interested in developing a mechanism that would notify employees of tasks requiring their attention separate from e-mails. His concept was to have a notification program installed on each employee's computer which would receive the critical notification information and display a pop-up message like Google's G-mail Notifier. [1]

We received a lot of good information during the meeting. We learned that the primary e-mail problem arises from a product called Patch Manager. This product is a web application written primarily in the Perl programming language. It manages Customer Support Tickets as well as the Patch Build Process. Each type of event has its own step-by-step procedure, which is followed every time there is a new ticket or patch. When one of the events needs to move onto the next step, Patch Manager generates an e-mail that notifies a predetermined group of people. This group of people has no control over what types of messages they receive, or how often they receive messages. Everybody receives messages when their attention is required in a ticket or patch, but they also receive other notifications that do not require their attention. The sheer number of e-mails received causes the important action messages to be lost. Paul made sure to mention that his team had been doing a lot of their new development in Ruby on Rails because it has a simple interface to implement web applications, and large scale applications are easier to understand than when done in Perl.

With this information we formulated a project proposal for a two-part software application that would solve the problems at hand. The first part is an e-mail filtering program. This filter system would give Cisco employees the ability to decide what e-mails they receive. However, the e-mail system still requires the employee to go to their e-mail and read the new messages to see when they have an action required in the Patch Manager. The second part is a notification mechanism. The notification mechanism would automatically display a message box on an employee's computer when that employee had a new action in the Patch Manager.

1.2: PLANS

Before we began working at Cisco's campus, our understanding of the current notification system was limited. We obtained our knowledge from a single in-person meeting with Bill and Paul in January 2006. The first step to working on this project is to understand the existing systems better. In order to gain this understanding we need:

- Access to the source code of the current system
- Access to any databases used by the current system
- What tasks are currently monitored by the system? A description of each step of these tasks would be beneficial to creating a flow control product.
- Who are the users of the system and what groups are they broken up into?

Once we understand the existing system we need to decide to either implement a new system to handle notification or to implement changes to the existing system. Either choice needs to meet the following three goals:

1. Streamline the current e-mails which are currently being sent out. This can be done by some combination of: removing users from tasks which have no relevance to their job; and/or, allowing users to opt in/out of notifications with no direct impact on them.

2. Create a push-forward notification system, which actively tells a user what tasks are still waiting for action.

3. Due to the nature of work done at Cisco, not all users use the same operating systems for their work. Therefore modifying the system so that it is implemented cross platform would allow inclusion of all employees at Cisco.

Our initial design idea is to break this project up into two parts, which will progress in parallel. The first part will be working on enhancements to the current system to reduce the number of e-mails which are sent. The second part will be implementing the push-forward system. Once the project is underway, it will be much easier to create a timeline of events. Once our timeline is established we can choose to either implement the two parts separately with add an adapter for communication, or we can develop both parts in a single application to increase efficiency.

There are many solutions to this problem; however there are two conditions put on the solution. First, we only have seven weeks to familiarize ourselves with the current system, design, and implement the new system. Second, whatever modifications or new software we put in place must be able to be modified and checked for bugs by current employees of Cisco. These requirements mean that we must use a development tool or language which is currently used by Cisco. Many members of Paul's team have been experimenting with development of applications in Ruby on Rails and they highly recommended using it for our task.

1.3: ROAD MAP

Cisco Systems has asked us to create an enhanced filter system to augment the one they current have implemented. The system we create can either be a derivative work of their current system or a new creation of our own design, however either with either choice, the system must be created using a programming language currently supported by full time staff of Cisco Systems. The new notification system must reduce the number of e-mail messages received by Cisco employees on a daily basis by allowing them to opt-in or opt-out of specific categories of messages. The new system must also have a push-forward service, which actively notifies users when there are pending tasks that require their attention to proceed to the next step in development. The goal for the new notification system is for it to work cross platform, with at least Windows and Unix environments. Our initial plan for development is to split the project into two parts. One part will work on developing the opt-in/opt-out system and integrating it into the current notification system. The second part will work on developing the push-forward system.

This report describes all of the work we have done while designing and creating the new notification system. It shows how we created our initial design to overcome the problems presented to us, and changed in order to fit the timeline and scope of our requirements. We familiarized ourselves with the existing system by analyzing what e-mails are sent by the Patch Manager system, and how it decides who receives those e-mails. We setup a temporarily web server to develop our application. After we had a server, a more detailed understanding of the problem at hand, and a firm design plan, we began implementing the project. Our implementation was split into two phases. The first phase was to get the basic functionality of an e-mail forwarding system working and a user interface to access the options of that system. The second phase added all of the features necessary to make the filtering mechanism operate and refined the interface to make it better fit user needs.

CHAPTER 2: PROBLEM STATEMENT

It appears that the current e-mail notification system was designed as a temporary notification vehicle and as the development process has improved the notification system has not been maintained in such a way that its efficiency persists.

2.1: E-MAIL SHORTCOMINGS

There are several software tools used by employees of Cisco in Boxborough. These tools have no native support for passing information from one to the other, or for alerting other employees when actions are needed by them, in order to advance the current project. The current system monitors the tools, and broadcasts messages whenever a project comes into a state where action by another employee is needed. However, these notifications have a broad scope. They are dispersed to more employees than just the person who needs to take the next step in the project. We have been asked to modify or replace the current system in order to meet the following needs:

- Reduce the number of e-mails which are sent – some users are receiving e-mails that have no bearing on their current job at Cisco. Going through the current list of tasks and verifying that all users or groups that are configured to receive e-mails for that task actually need to be informed about it.

- Allow users to customize the alerts they receive – some users may prefer to receive constant updates about the state of their work, while others may only be concerned with the actions they are required to perform. This is a matter of personal preference and should be controlled by the developers and managers themselves.

2.2: PUSH VERSUS PULL COMMUNICATION

In addition to the overwhelming users with e-mails, the current system lacks the ability to actively alert users about required actions. Since all communication is done with e-mails, users are required to check their e-mail and read each message to determine if they have a new item requiring their attention. This is pull communication: to get any information users have to go to a website and pull the important information out. Users can easily become distracted with other work, and may not realize they have a high priority action item for minutes or even hours after it was received.

Push communication is similar to pull in the way it is sent. The originating system passes its information to a server which then routes the information to the user. However, the delivery system is different. Instead of relying on the user to pull information, a push system receives the information and presents it to the user automatically. There are many ways a push system can be implemented. We were also asked to modify the existing system or develop a secondary application to accomplish the following task:

- Devise a method to actively inform users of outstanding action items – by eliminating the need for users to constantly check a web page or check e-mail, users will be able to focus on their work until they are notified of a new task.

2.3: SUMMARY

E-mail is a versatile tool for communication because it can be sent from any platform, from automated systems, and provides nearly instant delivery of messages. However, when overused e-mail can become cumbersome to the user when large amounts of information is passed at the same time, and no system for prioritizing the messages is implemented. Although this problem seems simple there are a number of possible solutions. In order to determine which solution would work best for our project we reviewed several existing projects that provide similar features. We also began

reviewing some tools we may need to use in order to implement our project.

CHAPTER 3: BACKGROUND

This chapter outlines related research that has been done regarding similar problems. It also covers tools that we think will be useful in solving the problem we have been presented with.

3.1: GMAIL NOTIFIER

Many people choose to use web-based email services, because of the ease of use, familiarity and ability to access it from nearly any computer with an internet connection. Since users can only discover new e-mails by actively checking their email account, rather than being notified automatically. A common solution to this is a notifier, such as Gmail Notifier [1]. These programs check for new e-mails intermittently and when one arrives, a noticeable notification is displayed in the corner of the user's screen, which is intended to be non-intrusive and disappears after a short delay. This approach overlaps with our project in that it bridges a gap between push and pull communication systems systems. The major advantage to this style of notification is that data is automatically pulled by a daemon on the user's end, which creates the effect that data is being pushed to the user without their having to "listen" for it or for a server to have to keep track of which systems are available to receive such notifications.

3.2: TESTTRACK PRO

TestTrack Pro is an application designed by Seapine Software that is used to track software defects. [2] This application on is a powerful tool as it can be used to not only keep a running list of the defects in the software that need fixing, but also to coordinating the efforts of many team members to ensure that the defects get fixed in a timely fashion. The assigning of responsibility to one or more users is analogous to the action item e-mail notification currently in place at Cisco. When changes are made, certain users must be notified so that the appropriate action may be taken

so development may continue. The TestTrack system sends out only the essential e-mails: users are notified by e-mail when they are assigned a defect to fix, and then assign the defect back to a build manager who is notified. The defect is then verified and marked ready for deployment, at which point, the person or group associated with final deployment is notified to include the fix in the next deployment.

Another useful feature of TestTrack Pro is that it is customizable. Administrators may add or remove categories of defects, customize groups of users who are notified when certain events occur, and so on. This allows the product to change with the needs of the company. For this project, the important lesson to be learned from TestTrack is that the notification system must be customizable to suit the changing needs of Cisco. We must not only develop a solution for now, but a solution for the future as well. Also, minimizing notifications to high priority items will allow users to quickly identify and begin work on action items and process them in a timely fashion.

3.3: RUBY ON RAILS

During a brief spike into Ruby on Rails [3], we have found that it is extraordinarily easy to get a basic database interface up and running. By simply plugging some information into the database configuration file and adding a few simple lines of code, add, edit, remove, and view functionality is available for a given database table. It is also not difficult to modify these pages to give a different look and feel, etc. The other nice thing feature of a Ruby on Rails solution is that it meets their requirement of being cross-platform as it is possible to run Ruby On Rails on Windows and Linux (our test environment was Fedora Core 4). Ruby also connects easily to any common database type. mySQL was mentioned specifically in the preliminary meeting and we have verified its functionality. Additionally, a Ruby On Rails solution will also provide a good educational experience for us (the team) and give Cisco a strong demonstration of the power of Ruby and what can be learned in a

short period of time. We will continue to investigate Ruby and explore more of the advanced features. Other solutions must be examined, but Ruby on Rails appears to be a viable option for this project.

Before we started the project we needed to learn how to implement web applications in rails. Rails has the ability to generate models, controllers, and views. After creating a Rails application it must be configured to connect to a database server and a specific database on that server. Once connected to the database, Rails can be used to generate a model. Rails automatically connects the model to a matching table in the database. The model is connected by looking for a table with a name that is the pluralized version of the model's name (i.e. the model 'person' would look for a table in the appropriate database named 'people'). Controllers and views are used to generate the actually web page a user would see when accessing the web server. The controller is written to preload variables and does calculations before the page is generated. The view is a HTML document with integrated Ruby code. When a client requests a web page, the web server runs the controller first to load the necessary variables, then opens the view, and begins passing the information in the view back to the client. While passing information to the client the web server is also interpreting the embedded Ruby code. There are two types of Ruby fields in a view and both take the form of HTML tags. '<% … %>' and '<%= … %>' are the two Ruby tags which can be used in a view. When the tag with the equals sign is used, the contents of the tag are interpreted and the results are passed to the client as part of the HTML document. When the other tag is called the code is executed, but nothing is passed to the client.

3.4: REALLY SIMPLE SYNDICATION

In order to solve the push vs. pull notification problem, a web feed version of the "open task list" could be created. RSS (Really Simple Syndication) is a standardized format for such feeds that is

based on XML [4]. From the viewpoint of the user, a feed is simply a document with a known URI that can be subscribed to by an aggregator (often referred to as a news reader). Several aggregators already exist that are both free and open source. On the server end, a feed can be generated by a script, that can be written in any number of languages, including Perl, PHP and Ruby. Since the current web-based system displays entries in an XML format, generating a feed would involve customizing that XML to match the standard. Each open task would be represented by an item in the feed, and would include links that perform the set of actions that exist on the current page (for example approve, deny, mark complete, and so on). Succeeding in this, we could then research aggregators to match the needs of the Cisco employees. Such an aggregator would automatically check the feed at pre-defined intervals and report any new items to the user through a system tray notification, balloon or Gmail Notifier style alert system. Clicking on the notification would open the aggregator, which would display the complete list of open tasks for the user to interact with. This type of solution seems like a good choice because it is based on an established, open standard and could be implemented in a variety of languages, including but not limited to Perl -- the language of the current code -- and Ruby.

3.5: LIGHT DIRECTORY ACCESS PROTOCOL

Light Directory Access Protocol (LDAP) was originally specified in RFC1487 [5]. It was intended to make read and write access to a user directory available remotely for use in simple management and browser applications. LDAP can be used as an authentication protocol, which uses the client-server model to centralize security for user and group access information. LDAP allows a web application to call the LDAP server with a user name and password and receive a message stating if that name and password combination is valid. If it is valid, LDAP makes other functions available. The LDAP server can be queried to see if a valid user is a member of specific groups.

LDAP can also be setup to store biographical about the user, so a web application could get the user's full name from a valid user name.

3.6: SUMMARY

The Gmail Notifier is a good example of what we want for a final solution. However, the Gmail Notifier is not open source and therefore we cannot extend it to meet our goals, but we can use it as a guideline while developing our new system. TestTrack Pro is a powerful piece of software, but without further understanding of the existing projects, we do not know how useful it will be. RSS is a communication standard that uses an automatic pull communication system that emulates push communication. The standard is based on eXtensible Markup Language (XML), which is a great way to pass data objects over the Internet. Creating an RSS feed is exactly like creating a dynamic website, except the feed creates an XML document instead of an HTML document. Many RSS readers have behavior similar to the Gmail Notifier so users get an automatic notification of new additions to the feed. There are also a few open source RSS readers available, which could be modified to fit any specific features we need.

Although we have a lot of research about similar solutions and development tools, we still do not have an understanding of the existing notification system. Paul asked us to analyze the e-mails sent by the Patch Manager. In particular he wanted us to discovery what types of e-mails it sends, and how it decides who each type of message is sent to.

CHAPTER 4: E-MAIL ANALYSIS

For this first stage of the project, we were tasked with examining the current e-mailing system of the Patch Manager and determining if there were certain recipients that did not need to receive certain types of e-mail and what e-mail types should be configurable. This is the first step towards reducing the volume of unnecessary e-mail. This analysis provides an organized representation of the recipients of the various e-mail types. This data is somewhat scattered throughout the Patch Manager, so this representation allows for easier decision making.

4.1: GATHERING THE DATA

In order to obtain the data for this analysis, some searching had to be done through the Patch Manager files. The basic procedure was as follows: open the e-mail template files to determine what variable populated the "TO:" field and then track the variable back throughout the application and determine how the variables were populated. Some of these variables were populated through queries to the database, which returned single or multiple e-mail addresses, while others were populated by server variables specified in the configuration files. Table 1 shows how each variable, in the e-mail templates, was populated.

Approvers General/Release	Determined by ProductID from the Approvers Table
Notifyees	Determined by TicketID from the PatchTicketNotifyees Table
Reviewers	Determined by TicketID from the PatchTicketReviewers Table
Ticket Users	Determined by TicketID from the PatchTicketUser Table
Build Team E-mail	Determined by ProductID from the ProductProperties Table
Patch Manager Admin	Determined by the ServerUniqueConfig variable PATCH_MANAGER_ADMIN_EMAIL
Ticket Requester	Determined by TicketID and Requester=1 from the PatchTicketUser Table
Manufacturing E-mail	Determined by the ServerUniqueConfig variable SERVER_MANUFACTURING_MAIL

Table 1 - Patch Manager Variable Associations

4.2: RESULTING DATA

Once this data was collected, the results were entered into a mySQL database for easy manipulation, querying, and display. Results are shown in Table 2. The Patch Manager has forty-eight message types. Each message type is sent to a specific set of employees at Cisco. Table 2 shows the list of message types, by the name of their template, along with the groups that message type is sent to and the subject of sent by that message type. Some message types are missing from Table 2, they are: bad_file_spec, build_faked, current_file_associations_changed, failed_packaging, found_skipfiles, inherited_file_associations_changed, invalid_components, modified_after_merge, no_rne_found, no_xml_file, packaged, packaging_failed, patch_field_edited, re_open_parent, redirect, sr_created, submit_clarification, submit_defined, submit_denied. The missing message types are all sent to the "All Approvers" group. The "All Approvers" group gets the most messages from the Patch Manager and most of the messages they receive are not necessary for their daily work.

Template Name	Recipients	Subject
add_lrp_to_rup	Ticket Users	Action required: merging engineering special to service release
add_product_failure	Product Level Approvers	Failed to add Product to Patch Manager
approved	Ticket Requester	Patch request approved
bad_xpack_call	Patch Manager Admins	FAILURE: Bad SOAP call from externalpackaging system
bad_xpack_call_noobj	Patch Manager Admins	FAILURE: Bad SOAP call from external packaging system
build_complete	Ticket Users	Patch ready for file selection
cancelled	All Approvers, Ticket Reviewers	Patch request cancelled
code_review_complete	All Approvers, Ticket Reviewers	Code review complete
code_review_overridden	All Approvers, Ticket Reviewers	Code review complete (overridden)
create_build_repository_dir_failed	Patch Manager Admins	created_build_repository_dir_failed_test
denied	Ticket Users	Patch request denied
good	All Approvers, Ticket Reviewers	Patch queued for posting to CCO
marked_bad	All Approver, Manufacturing	E-mail Patch removed from CCO
more_info Ticket	Requester	Patch request requires more information
no_xml_in_build	Build Team	Bad or missing xml file in build
notify_users	Predefined Recipients	Predefined Subject
patch_reviewed	All Approvers Ticket Reviewers	Patch reviewed

patch_test_complete	All Approvers Ticket Reviewers	Patch Test Success
patch_test_failed	All Approvers Ticket Reviewers	Patch Test Failed
posted_to_cco	All Approvers Ticket Reviewers Ticket Notifyees	Patch posted to CCO
posted_to_cco_build_team	Build Team	Patch posted to CCO
re_open_child	All Approvers All Ticket Users	Patch request reopened
re_open_for_file_selection	Ticket Users	Patch request reopened
re_open_lrp	All Approvers All Ticket Users	Patch request reopened
ready_to_build	Build Team	Patch ready to build
ready_to_code_review	All Approvers Ticket Reviewers	Patch request ready for code review
review_failed	All Approvers Ticket Reviewers	Patch failed review
sample_readme_created	Previously Defined Recipients	Sample readme created
update_sitebranch_failure	Patch Manager Admins	Failed to Update Site Branches

Table 2- Message Types

This data allows managers to determine if there are e-mails that do not need to be sent to certain groups, assess which groups need permissions to other messages, which groups should be required to receive certain messages, and which groups are required to receive notification for certain messages.

4.3: SUMMARY

The Patch Manager is implemented in Perl and has dozens of source files. Once we found the function responsible for generating e-mails we discovered the Patch Manager has predefined templates for each type of e-mail it sends. Each template has a variable that defines the users e-mails of that template will be sent to. We compiled a list of recipient variables, sorted by message type, and traced each variable back to the database query that populates the variable with user names.

This process gave us a detailed look at the operations of the Patch Manager's notification system. We decided to attempt to emulate this system in the initial design of our new mail filter application.

CHAPTER 5: DESIGN

After completing the e-mail analysis of the Patch Manager we returned to our design concept which we had outlined in our project proposal. We had several new ideas that we felt would make the program more robust and could create a centralized communication system.

5.1: DESIGN CONCEPT

Having a central location to control user subscriptions to notification messages has several advantages. One such advantage is that when a new employee needs to be added to the system he/she can be added to multiple projects in one place. Also, when a user wants to configure their delivery settings, the system does not have to query subscription information in other projects to determine what the user currently has access to.

THE DATABASE

We decided to start on the design for the subscription process from the ground up. Our ground level is a mySQL database. We determined what data the service would need to know in order to function correctly, and then we decided how that data would be stored in a database. We used our knowledge of the Patch Manager's operations to start building the database. First the system would need to know details about each project it handles. So there would need to be a project table, which stores the name of the projects, like the Patch Manager, and a generated unique identification number for each project. The system would also need to know what types of messages each project can send. Message types became our second table, with a variety of fields. The two primary fields are the unique identifying number and the ID of the project the message type is associated with. The table would also store the name of the message type (as used by its associated project), and humanized named (for display to the user), a description (so a user understands the intent of the

message type), and general settings for the message type (like 'no block', which would prevent users from blocking delivery of that message type).

With these two tables it is easy to describe the basic functionality of the Patch Manager. However there is no way to store subscription information. In the Patch Manager, messages are sent to different groups based on the message type and the patch or ticket the message concerns.

The following is a possible situation which could arise in the Patch Manager. Patch 32 is started and three groups are created for that ticket: requestors, approvers, and programmers. Each of these groups has access to messages of specific types that concern Patch 32, like the requestors group has access to build complete messages for Patch 32. Twenty minutes later Patch 33 is started and the same three groups are created. However, for Patch 33 the requestors group does not have the same members as the requestors group for Patch 32. The new requestors group would still have access to build complete messages, but for Patch 33 instead of 32.

In order to emulate this behavior we created a group types table. This table contains a list of the available group types and what messages those group types have access to. So a requestor group type could be created with access to build complete messages. Now there needs to be a way to specify groups and their members. Another table was created that stores the group type identifier, the subscript for the group, and the names of the users in that group. So you could say that requestor group 32 contains user one and user two, and then requestor group 33 contains user one and user three. This allows the program to know that when a build complete message is sent from Patch 33, the requestor group for Patch 33 has access to that message, and that group contains user one and user three.

In this system we can also start with a user name to run a query and find a list of group types that

user is in. From the list of group types, each group type can be looked up and a list of message types can be generated which shows what message types the user has access to. This list can be presented to the user, instead of a list of all available message types, when he/she wants to set delivery preferences. The final necessary table is the user preferences table. This table contains a list of user names, message types, and delivery options. One entry could be user one, build complete, and block. So whenever a build complete message is sent a group with user one as a member, the message is not sent to user one.

This type of system requires input by the system administrator to define each project and message type. On top of that, the administrator also has to predefine the list of group types and what messages those group types have access to. The associated project then has the ability to create groups of the given types, add users to those groups, and send messages of the given types. Once a group is created and users are added to it, the project has no need to retain that information, because it could send a build complete message for Patch 33 and the subscription service has all the necessary information to determine who is supposed to receive that message.

REMOTE CALLS

This system requires a substantial amount effort to implement correctly. Also, there needs to be lots of changes to the existing projects to adapt them to the new subscription system. However, the subscription system could then be used, in future projects, as an easy to implement notification system.

In order for other projects to work with the subscription/notification service they need to have access to a variety of commands. An introductory list of commands includes:

- Create Group (called with a group type and identifying number)

- Add user to group (called with a user name, group type, and identifying number)

- Send message (called with a message type, message body, and identifying number)

With these three commands the entire system can be implemented. Once each of these commands is called, the rest of the system is basically a sequence of database queries. The only command that requires action outside the database is 'Send Message' because it needs to compose e-mails and send them to an SMTP server as well as add information to the necessary RSS feeds.

We discussed the subscription process with Paul and went over the design details with him. Paul felt that one system handling subscriptions for every Cisco application would become cumbersome quickly. Also, changing the existing systems would require more time than it would save. Although this decision took a large cut out of the projected work for our system, it gave us extra time in the remaining weeks to correctly implement the other features, and do some polishing of the entire system.

5.2: FINAL DESIGN

After removing the subscription service from our project, we were asked to brainstorm a list of features out filtering system would have, and to prioritize those features in the order they should be implemented. We came up with five primary features and list of questions we needed answered before we could start implementing any of the features.

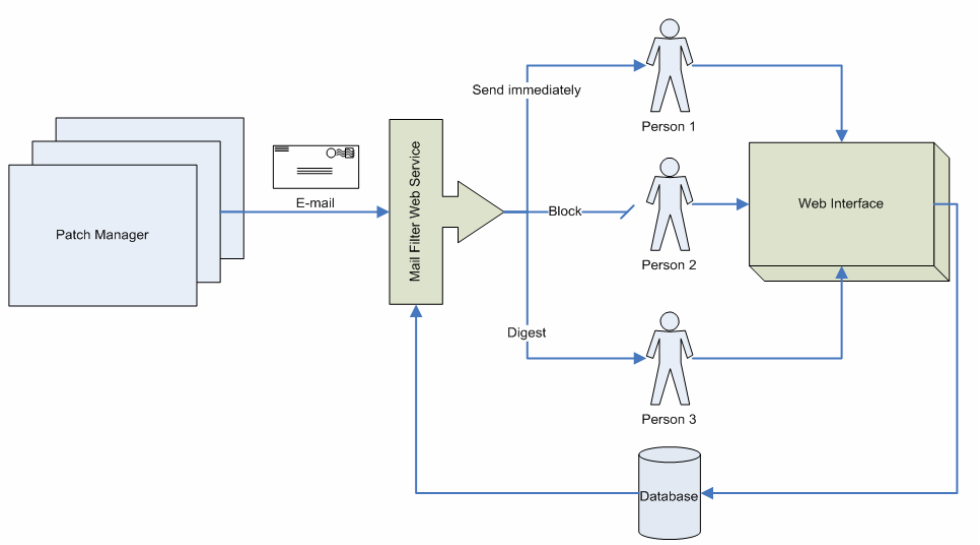

Figure 1 - Final Design Overview

FEATURES AND PRIORITY

1. **Send Mail** – Mail Filter will receive information from other applications, such as Patch Manager, as shown in Figure 1. Communication is done in a predefined XML Schema. The information received will be parsed, compared against user settings, compiled into a standard e-mail format and sent via an SMTP server.

    a. **Configuration Link** – Every e-mail sent by the Mail Filter will end in a link which tells the user where they need to go to change their Filter settings.

2. **Configurable Features** –

    a. **Project IDs** – Allow the administrator to setup groups of messages that are sorted by the Project where they originate from. This approach allows for multiple applications to use messages of the same name with different descriptions or uses.

Information about projects and messages is stored in the database shown at the bottom of Figure 1.

b. **Block Messages** – Allow users to see what messages they are currently getting and change a setting which would block them from receiving that type of message again.

3. **Notification** –

   a. **Indicate By User** – The XML Schema includes a listing mechanism to show the recipients of the information being passed. There will also be a way to specify if any subset, of recipient, is required to perform some action on the information being passed.

   b. **Notification Handler** – Mail Filter will have a secondary process which is responsible for pushing the information to users whose attention is required.

4. **Bulk Send** – In addition to allowing users the ability to completely block receipt of any message types, Mail Filter will also be designed to maintain a digest of certain message types and send them to the user in a single e-mail at a specified time interval.

In addition to prioritizing the features we also redesigned the database layout and came up with a basic flow diagram for the filer system as shown in Figure 2.

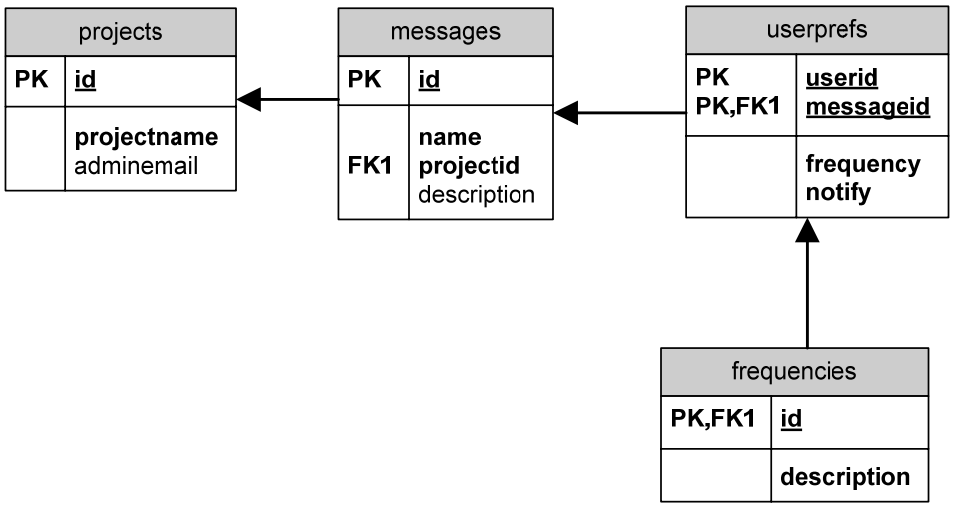

Figure 2 - Filter System Database Schema

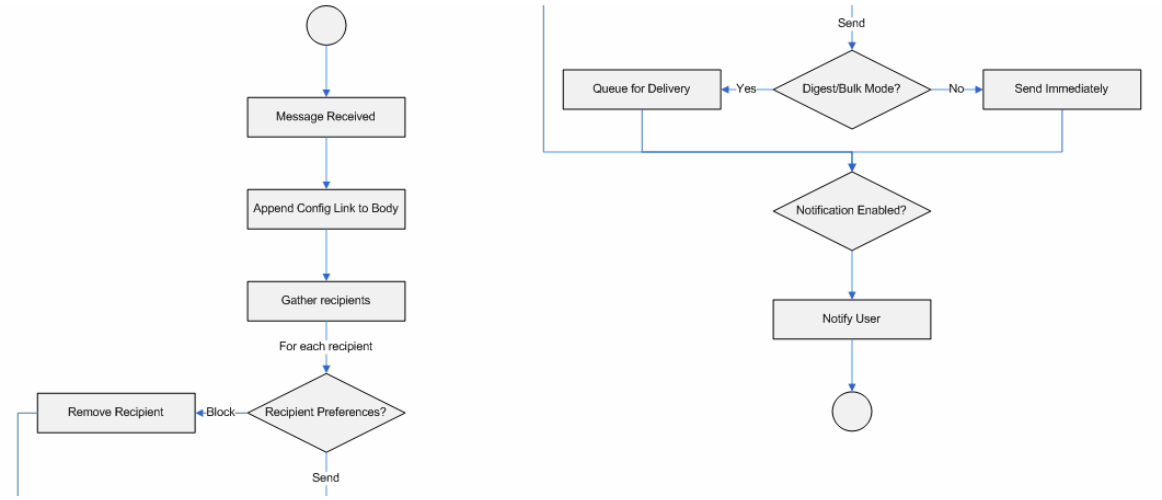

Figure 3 - Filter System Flow Diagram

OTHER DETAILS

In the filter system flow diagram, shown in Figure 3, there are a couple of places where the direction of the flow depends on a predefined user preference. Since the system does not know every user's preferences before it starts sending e-mails, we decided to give the system some default

behavior. The system assumes the least restrictive setting for a user preference when one is not defined. Such as, when the system receives a new message to send, and the user has no setting for that message type, the system assumes the user has that message type set to "Always" and sends the message immediately.

The system also needed some form of error handling. We want the system to be able to continue to function whenever there is an error without immediate intervention from a person. To implement fault tolerance we need to design the program to catch all errors, log the errors, and then report the errors to an administrator. The only errors we could initially think of are invalid message type and invalid project name. Since neither of these errors cause harm to the system, we decided to allow the system to still send messages with invalid types and project names. However, these messages are recorded and an administrator is notified so he/she can block those messages in the future, add them to the system, or fix the source of the invalid messages.

We also had concerns about how to pass action required information to the filter system from other projects. Currently when a system, like the Patch Manager, sends a message, there is no way to tie that message to previously sent messages in order to create a thread of events. When a message is sent saying a patch is awaiting approval and then another message is sent saying that the same patch has been approved, there is not a dynamic way for the filter system to know that the two messages are related. Without this relation we would be unable to maintain the correct state in an RSS feed. We want the ability to be able to add an action, like approval required, to the feed, and then remove it, as soon as approval has been given or denied. Paul suggested we split the project into two systems. The first being the filter system which would be solely responsible for handling e-mail traffic. The second would be the Patch Manager Notification system. Paul suggested we focus on the filter system at first, since the Patch Manager Notification system is a low priority right now, and one member of his

team would likely be able to help us set up an RSS feed inside the Patch Manager in a matter of minutes. Paul asked to see a Specifications Worksheet for the e-mail filter. This document, now referred to as Appendix A, was written and delivered on March 30, 2006.

Our final concern was about user authentication. We decided to use the light directory access protocol (LDAP) to authenticate users who want to log into the mail filter system and change their user preferences. Ruby does not have a standard library for LDAP so one of our immediate tasks would be to learn how to use the protocol in Ruby.

5.3: SUMMARY

Our final design is a complex Ruby on Rails application. In order to develop such an application we would need a web server capable of hosting Ruby on Rails and has the ability to incorporate other languages if necessary. We decided to use the Apache web server because Paul's team was already familiar with it, and it has the versatility we need.

CHAPTER 6: SERVER CONFIGURATION

As a team we defined the basic features we knew would be needed to have a successful web application running on our system: a web server, a database, and the ability to host common gateway interface scripts.

6.1: THE WEB SERVER

Paul's team currently uses an Apache Web Server running from a Windows machines to host their projects, so we decided to use Apache as our web server to be as compatible as possible with the existing Cisco setup. Getting Apache installed for Windows is easy. There is an Apache 2.0.55 One-Click installer for Win32 systems available from http://httpd.apache.org/. Although the current version of Apache is 2.2.0, there is not a Win32 installer available for that version. We did not foresee any compatibility problems arising from using the second highest version, so we used the existing installer instead of compiling a copy of Apache 2.2.0. We found a good walkthrough for installing Apache with various add-ons at http://mpcon.org/apacheguide/prerequisites.php.

6.2: THE DATABASE

The current version of the Patch Manager runs on an SQL Server database. However, we decided to go with a mySQL database instead. We felt that MySQL is easier to develop on because it is free software and easy to implement across platforms. Another reason we decided not to use SQL Server is because the SQL Server database is case sensitive and Ruby is not case sensitive which would make some interaction between the two systems difficult. Like Apache Server, MySQL also has a One-Click installer for Win32 systems.

6.3: CGI SCRIPT HANDLERS

Common Gateway Interface (CGI) scripts come in a variety of forms today. The most common is the Perl programming language. Although we had no intention of writing any code in Perl we decided to install a Perl interpreter anyway so that we could use or test segments of the Patch Manager code on our server. Perl comes in many varieties. The two for Windows systems are called ActivePerl and PXPerl. After doing some quick research on the Internet, we decided that ActivePerl was the better supported and most compatible with other modules we may want to use in the future. ActivePerl has a One-Click installer and seamlessly integrated into the Apache Server by adding one line, "AddHander cgi-script .pl .cgi" to the Apache configuration file.

Since we are running a mySQL server, we decided that having the ability to remotely administrate the server would be beneficial. The easiest way to do this is through a program called PHPMyAdmin. PHPMyAdmin is a web interface built on the PHP programming language. Installing the PHP interpreter was a lot easier than Perl because there is only one Windows distribution of PHP. After running the PHP One-Click installer, we modified the Apache configuration line from Perl to be, "AddHandler cgi-script .pl .cgi .php" in order to incorporate PHP to the web server. PHPMyAdmin is a set of PHP scripts put together to create a graphical web-based interface to a MySQL server. We added the PHPMyAdmin scripts to the public directory in the Apache web server, and made some configuration changes to the PHP installation. We had to set a line in the PHPMyAdmin configuration file to read:

- "$cfg['PmaAbsoluteUri'] = 'http://wpifilter/phpmyadmin/';"

This line only allows computers on the Cisco network to access PHPMyAdmin on our server, since the 'wpifilter' address will not resolve to a valid IP address outside the network. We also added the lines:

- “$cfg['Servers'][$i]['extension'] = 'mysqli';”
- “$cfg['Servers'][$i]['auth_type'] = 'http';”

These lines told PHPMyAdmin it would be connecting to a mySQL database and it should use the standard HTTP authentication supported by the Apache server. PHP also needed a couple of configuration changes to make PHPMyAdmin fully functional:

- extension=php_mbstring.dll
- extension=php_mysqli.dll
- mysqli.default_host = localhost
- mysqli.default_user = root

These lines tell PHP to connect to a mySQL server on the local machine using the mySQL user name ‘root’ for accessing the database by default.

6.4: RUBY ON RAILS

Although Ruby is another interpreted language which can be used for CGI scripting, Ruby on Rails requires a lot more effort to integrate into Apache than Perl or PHP. Installing the Ruby interpreter was much like Perl and PHP in the fact that there is a One-Click installer for Windows Systems. Ruby comes with a program called RubyGems which is a modules installer for Ruby. RubyGems looks up module packages at http://rubyforge.org/, downloads the designated package, and installs it on the system. Once the Ruby interpreter and RubyGems are installed, RubyGems is used to install the Rails package and all of the packages it depends on.

Hosting a Ruby on Rails application with Apache uses the command line Ruby interpreter to process the various Ruby files and produce HTML every time a user connects the application. However, the command line interpreter recompiles all of the Ruby code every time it is run, which makes it a very slow implementation for hosting complex web sites.

6.5: FAST CGI

FastCGI is a language independent, scalable, open extension to CGI that provides high performance without the limitations of server specific APIs (http://www.fastcgi.com/). The FastCGI module for Ruby allows Ruby processes to be spawned in the background of the web server with preallocated resources. The spawned processes have a much better response time than Apache's standard CGI implementation.

Installing FastCGI does not take a lot of time. Adding the following line to Apache's configuration file is the first step:

- "LoadModule fastcgi_module modules/mod_fastcgi.dll"

The next step is to specify the Rails applications which will need to be spawned and how many of them to spawn:

- "FastCgiServer /path/to/rails/app/public/dispatch.fcgi –processes 1"

The number one at the end of the line specifies the number of processes to spawn when Apache is started. This number needs to be balanced between the available resources of the server and the number of connections that are expected for the application. For testing purposes we use one process.

The final necessary change is to the ".htaccess" file in the public directory of the Rails application being hosted. The 'cgi' on the following line is changed to fcgi:

- "RewriteRule ^(.*)$ /myapp/dispatch.cgi [QSA,L]"

This line tells all requests sent to the Rails application to be passed directly to one of the running processes.

6.5: SUMMARY

Our web server has the ability to host applications in Perl, PHP, and Ruby on Rails. Each language can be processed by Apache's default CGI module, or through the FastCGI module we added. The server also has a local mySQL database so all of the languages have a central data storage mechanism.

With these tools ready, we began developing our system. Since we started implementing the project while the design process was still underway, our implementation went under several changes when the design was altered.

CHAPTER 7: IMPLEMENTATION

After the subscription service idea was changed to a simpler system, we started implementing the basic functionality for the mail filter system. The Cisco E-mail Filter System's two primary functions are to receive the standard fields for an e-mail and then to send that e-mail to its designated recipients. Both of these tasks were easily accomplished with the Ruby modules Action Mailer and Action Web Service.

7.1: ACTIONMAILER

Since the Mail Filter is responsible for actually delivering the e-mail, a mechanism to send this e-mail is necessary. Ruby on Rails has such a mechanism, which is called ActionMailer. ActionMailer creates TMail objects in which the e-mail headers (to, from, cc, bcc, reply-to, subject, body, and so on) are defined and then sent.

For this application, the ActionMailer class contains functions for three types of e-mail: general_email, admin_email, and unfiltered_email. The function general_email is used for most e-mails and passes the recipients through the filter to determine how they wish to receive messages. The admin_email is sent to the administrator of a given project typically to notify the administrator of an unrecognized message type. Once the administrator adds the message type, users are able to set their preferences. Finally, unfiltered_email is used for actually sending bulk e-mails. This function can also be used to send e-mails that users should not have the option to block.

7.2: ACTION WEBSERVICE

An essential part of this project is a web service which accepts calls from the applications whose e-mails will be filtered. Ideally, the web service could be a part of the Ruby application so it could leverage all of the models and functions of our existing application. This approach is possible

through the use of ActionWebService. This library simplifies the construction of a web service to a mere few lines of code for basic functionality. A web service created with ActionWebService consists of two distinct parts: the API and the web service controller. The API is a simple definition of the functions provided by the web service including expected data types for each of the parameters, a list of function names, and expected return types. The API is flexible, especially prior to the implementation of calls to the web service. Fields can be added and removed with minor changes to the code. The web service controller, like all ruby controllers, handles the routing and processing of requests.

The API for this web service is simple. The web service has one function called SendMessage, which takes normal e-mail header fields as parameters. Additionally, the project and message type must be specified in order to properly identify messages for filtering. Using project IDs and message types allows flexibility, as messages are not bound to any particular subject or body in order to be identified correctly.

The web service controller handles the actual processing of the incoming message. The flow of the web service controller is shown in Figure 4. The type and project of the incoming message are checked to ensure that they are valid. If this information is invalid, the e-mail is sent, but the project administrators are notified of the failure. Assuming the message type and project are valid, the incoming message has its recipients passed through the filter to remove any recipients who have opted to block or receive digests of the incoming message type. Digest users have entries created in the database so a script that runs periodically is able to send the messages at the appropriate time. Once the users are filtered, the custom signatures are added to the bottom of the e-mail body and the message is sent to those users who elected to receive e-mails immediately.

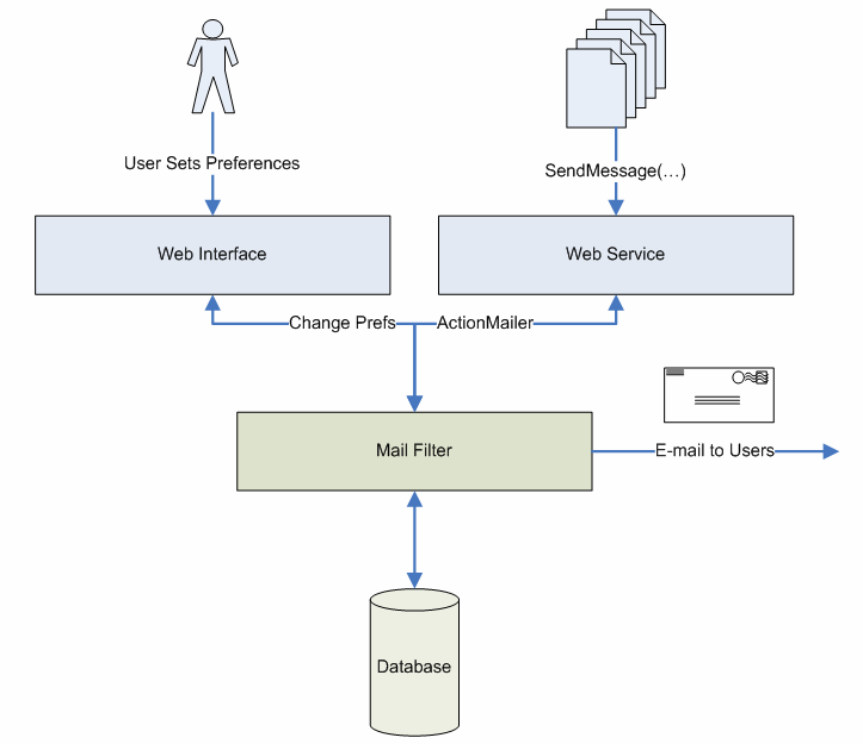

Figure 4 - Web service Flow Chart

Unfortunately, threading in Ruby on Rails seems somewhat unstable and caused mySQL errors when we attempted to make the web service itself multithreaded, so for this reason, the web service verifies the project ID and message type then stores the message into the database and returns a status message to the calling project. Another script monitors the database and sends any new messages. Storing the messages in the database and sending them with another script makes the web service to be asynchronous and allows the calling applications to continue their normal operations without waiting for Action Mailer to send every e-mail.

The web service was expanded to support blocking of e-mails and adding appropriate rows to the user preferences table. These two functions are essential to the operation of the Mail Filter application and will represent the bulk of the useful functionality.

The block functionality is the part of the web service that actually interprets the list of recipients from the calling application and removes users who have set their preferences to block. This functionality is accomplished by querying the user preferences for users who have set preferences for the given message and have set their frequency to "Never". The web service parses the comma-separated list passed in the "To" and "CC" fields and removes the appropriate users.

In order to determine which messages a user should be able to configure, we elected to only allow users to configure their preferences for messages they have already received. This way they will only be able to see and customize settings that are relevant to them. This also ensures that users receive at least the first e-mail of any given type, so they are aware of their settings for each message type. For this reason, the web service checks if the incoming message is of a recognized type and if so, checks if a user has existing preferences for this type. If so, the appropriate action is taken (remove the recipient if "Never" or store the message if "Bulk is selected"). If not, the message is sent to the recipient and an entry is created for the user and message in the user preferences that is by default set to "Always" receive the message. When the user logs into the User Console, he will now see this message in the list of customizable messages.

7.3: USER INTERFACE

Our user interface started out very simple, shown in Figure 5, and underwent a major overhaul after some suggestions from Paul. Our original layout contained tables that were simply a representation of the database contents in the same format. This format was reasonable for short fields such as frequency, but was problematic for fields like description, which could be quite lengthy. This format also did not conform well to high resolutions or small amounts of data.

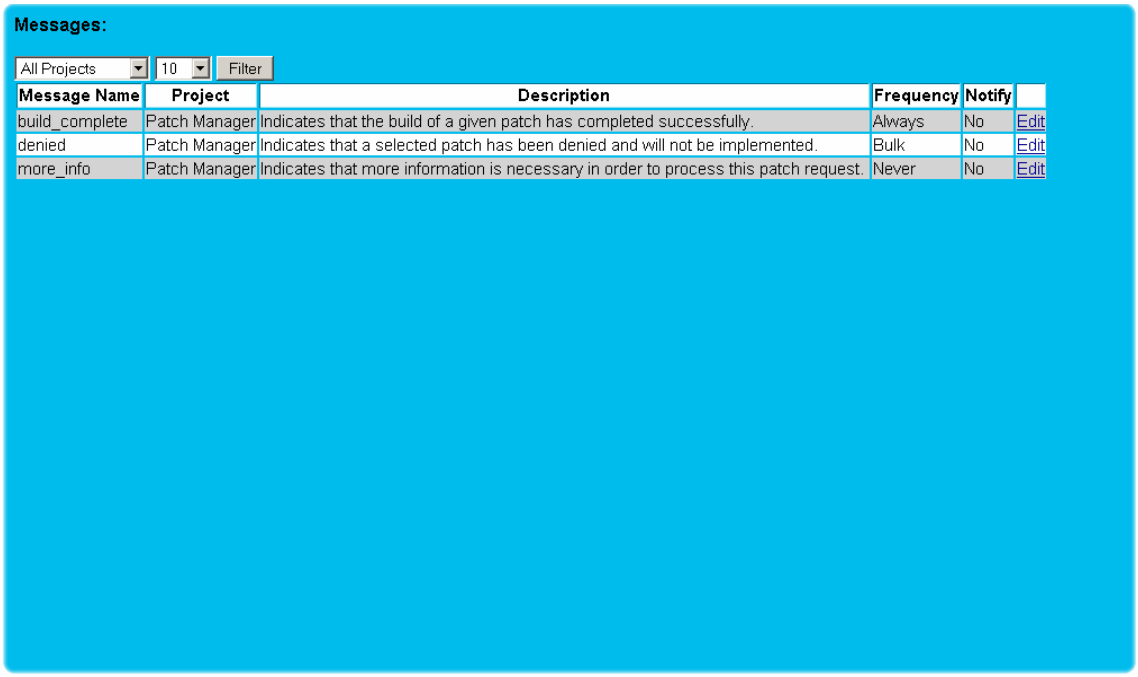

Figure 5 - User Interface First Draft

This view would have been practical for large amounts of data and small fields, but in many cases this was not the type of data that was displayed.

A meeting with Paul revealed problems with the existing user interface. Paul pointed out the following problems with the above user interface:

- Data was constricted to small regions of the page, which did not properly leverage larger displays or display small amounts of data in a useful way.

- The framing of the page was nice, but a minimalist approach is more pleasing.

- Some sort of navigation needed to be implemented to allow administrative users to switch between administration pages and their own custom preference pages.

- He suggested we use an interface similar to that of http://www.digg.com/. This format leverages different resolutions better and allows large fields and small amounts of data to be displayed efficiently.

We agreed with all of his observations, and devised the interface below to meet these needs. This format looks far better on different resolutions and represents small amounts of data and large fields as suggested. We had another meeting to get feedback from Paul's team on this user interface, but we already feel this is a far better approach. The new interface, show in Figure 6, shows the same information as the original interface, and makes editing several options available on the same page instead of nesting the options on other pages.

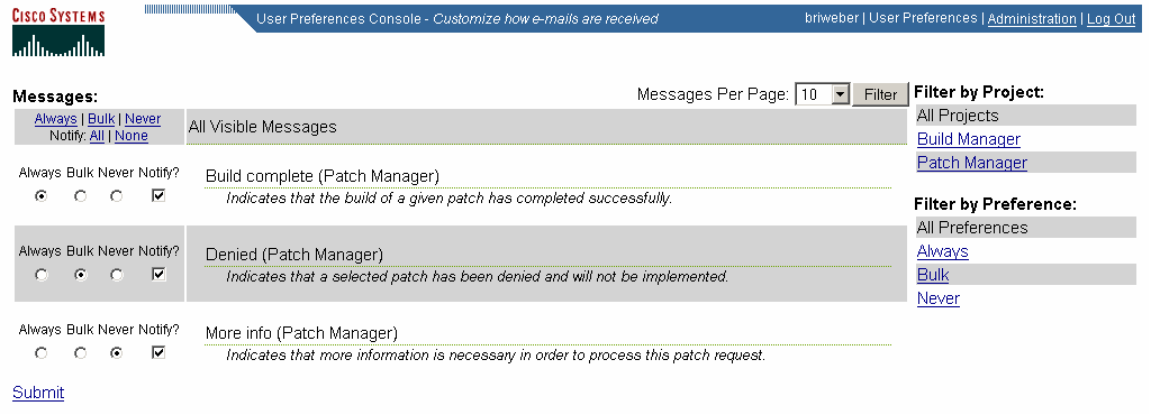

Figure 6 - Final User Interface

After the second meeting for feedback on the user interface we came up with a list of additional ideas to incorporate into the system.

- Instead of clicking a button and going to a screen to edit preferences individually, put radio buttons on the console page so users can change quickly.

- Link to web service information in the Administrator console. Show sample code for each

programming language used by Cisco.

- Add a select/deselect all so that users can change all of their settings for a project at the same time.

- Provide the title of the page and basic instructions to what the page does.

- Change the Administrator system so that there are Super Administrators who have access to the entire system and administrators designated for specific projects. These administrators should have the ability to create and remove administrators at their level as well as edit global settings (such as project signature) for the projects they have access to.

- Allow users to filter their messages by current preference

These recommendations were added to our project design. As we added more functionality to the implementation of the mail filter we also implemented these recommendations.

7.4: LIGHT DIRECTORY ACCESS PROTOCOL

With the user interface implemented we need a way for the interface to know whose options it should be displaying. This is done by having the user log in. Any log in function requires there to be some sort of authorization so users cannot log in as someone else and change their preferences. Cisco already uses LDAP on their existing systems so we decided to extend there implementation for the Cisco E-Mail Filter System.

LDAP IN RUBY

At first LDAP in Ruby on Rails appeared easy to use. Ruby on Rails has a RubyGems module called Ruby-ActiveLDAP. ActiveLDAP is a Ruby object with fields for configuring the connection

information and passing a user name and password. The object has its own functions for binding to the LDAP server, verifying a user name and password combination, getting the user's biographical data, and checking the user's group memberships.

The RubyGem ActiveLDAP module requires the Ruby-LDAP extension library. The library is a compilation of source code written primarily in the C programming language for Unix systems. The result of compiling the source code is a function library which can be loaded into the Ruby interpreter. Since we are running all of our systems on Windows, the Unix based source code was not helpful. An hour of research revealed Ruby enthusiasts had found a way to compile the source code under Windows with the Cygwin tool, available at http://www.cygwin.com/.

Cygwin loads a Unix command shell on the Windows environment and translates Unix system calls into the corresponding Windows system calls. Cygwin installs without a problem, and then presents an enormous list of modules which can be installed. By default none of the modules are selected. Cygwin is setup to recognize the Windows path and will execute both Unix based tools as well as Windows tools from the Unix command shell.

Ruby-LDAP comes with a Ruby script to check system configuration options and generate a Makefile which can then be used to compile the C source code. Running the script with Cygwin returns an error about missing header files, "ldap.h" and "lber.h". More research reveals these files are available as part of the OpenLDAP project. OpenLDAP is another compilation of C source code that implements LDAP functions. The script has command line options to direct it to the LDAP source code directory. After several attempts, it was discovered that Cygwin was using the Windows installation of the Ruby interpreter which told the script to use a Windows compiler for the C source code. Of course, Windows does not come with a C compiler. However, Microsoft does offer a free

C/C++ compiler now. With this installed, the Ruby script successfully created the first Makefile.

The first Makefile was riddled with problems, primarily the fact it was trying to use a Windows C compiler in the Unix environment under Cygwin. To remedy this situation, we went back to the Cygwin module loading screen and added the Ruby Module, gcc (Unix C and C++ compilers), and make modules in order to add full native Unix support for Ruby and C. Running everything again from a clean environment produced a new Makefile. When run, the newest Makefile complained about not being able to find other LDAP header files. These files are clearly located in the same directory as "ldap.h" and "lber.h", which are both found successfully. No quick solution was found for this problem, so Ruby LDAP was dropped temporarily and we moved on to other aspects of the project.

We renewed our efforts towards getting LDAP to work with Ruby when we discovered and LDAP source library file in the Cygwin modules. With this module and a few dependencies installed, everything built fine. We had our first copy of the library module.

Modules are easy to add to Ruby, they are put into a directory in Ruby's include path and then used with a "require" line in the Ruby source code. There were some basic problems getting ActiveLDAP setup, but they were quickly overcome. Once everything was setup, we wrote some test functions to see if basic LDAP functionality existed. Hours of debugging later, and we had discovered just using the "require 'activeldap' " line was causing an error in our application because the Cygwin compiled version of Ruby LDAP does not work under the Windows Apache environment.

LDAP IN PERL

Since LDAP with Ruby on Windows seemed to be getting further away instead of closer, we

dropped the idea and switched modes to Perl. Perl was already installed and configured on our web server, and Perl was used to implement LDAP for the Patch Manager. Copying the settings from the existing LDAP implementation in Patch Manager took a little bit of time, because some components were integrated into the Patch Manager, and would need to be extracted before they could be used in the filter system.

The way we currently have LDAP implemented a client can connect to our server and is immediately prompted for a user name and password. Those items are validated with LDAP and then the user is allowed into the Ruby application. Using the Perl Module for Apache, the web server is configured to run the designated authentication script in Perl whenever a user attempts to log into the system. This script is the piece we extracted from the Patch Manager which connects to the existing Cisco LDAP server for user and password validation. The script is currently setup to store the user name into an environmental variable, which can then be accessed by other Perl scripts under the same connection from the client.

Storing the user name into an environmental variable worked when using Ruby on Rails in normal CGI, however, when we switched to FastCGI for performance the LDAP system broke. When an application is run under FastCGI, it creates its own environment and does not have access to the environmental variables from other CGI applications on the same server. Using environmental variables to pass the user name would not work, but we knew we could not get LDAP working in Ruby so we had to find another way to get the authentication information into the Ruby application.

FastCGI has a number of functions which can be called from the Apache configuration file. The most important one for LDAP is pass-headers. When a FastCGI application is defined in the Apache Configuration file, "-pass-header Authorization" can be added to the end of the line, and Apache will automatically add the appropriate authorization information to the new FastCGI environment. This

allowed us to directly access the user name from within Ruby.

7.5: RAILSCRON

In order to make the Web Service asynchronous, when it is called, the function simply validates the message type and enters the data into the database for later processing. In order to actually process these messages, we made a script that extracts unsent message data from the database, filters the recipients, sends the e-mail to recipients who receive it immediately, and queues it for digest delivery for those users who choose that method.

This task can be accomplished in many ways, including Windows Scheduled Tasks or threading for example. Windows scheduled tasks are not customizable and can only be run at 24 hour intervals or greater. Threading seems somewhat problematic in RubyOnRails since it is currently more of an implemented hack than a feature.

The alternative is RailsCron. This is a RubyOnRails plug-in that allows functions to be scheduled to run between two specified times at a specified interval. This approach is convenient because these tasks can be manipulated by the application itself and verification can be performed to ensure the tasks exist.

Overall, this approach is straightforward, but in order for the scheduled tasks to run, they must be started. Once started, the RailsCron instance polls the database to gather information about the tasks to perform. In order for this functionality to work properly, we created a Windows service to start the RailsCron instance. This approach has a number of advantages:

1. Windows services are started even before a user logs into the system and will run automatically after a system restart.

2. There is no command prompt window visible to clutter the server.

3. If the RailsCron instance crashes, it will be restarted automatically by the Windows service.

Tasks are created for both the mail processor and the bulk message e-mailer. The mail processor runs once a minute and processes all unprocessed messages in the database. The bulk message e-mailer runs every hour and sends digests e-mails for all users who have chosen that preference. Customizability for bulk send intervals has not yet been implemented, but would be a valuable feature to implement.

7.6: SUMMARY

Creating the tools to link Ruby's ActionMailer and ActionWebService has half of our implementation. In the end we decided to use two separate scripts for each task. When the webservice is called one script adds the data passed to a database. A second script, running on RailsCron, monitors that database for new information, assembles that information it into e-mails, and passes the e-mails to the ActionMailer mechanism.

We then added the user interfaces to allow users and administrators to configure options for e-mail delivery and project setup. The LDAP script we used was a slightly modified version of the script used by the Patch Manager system which gives our system the same level of security used by other Cisco products.

The only missing feature in our implantation is the notification. RSS provides an excellent tool for notification because it is an existing standard, and meets our cross platform requirement. There are RSS readers available for every platform and many readers are available on multiple platforms.

CHAPTER 8: REALLY SIMPLE SYNDICATION

Our notification system was intended to create an Really Simple Syndication (RSS) feed, which could be subscribed to by Cisco employees and would generate a feed for each of them. The feeds would be a list of tasks that need to be accomplished by the subscribed user. RSS seemed to be a good way to do this because it would save the effort of creating a new communication and notification protocol.

RSS is an increasingly popular way to notify users around the world about events in almost real time. RSS is constructed by two separate pieces of software, a feed and a reader. The feed is a file or piece of software that makes a defined set of information available to a group of users. The reader is a piece of software which connects to the feed, downloads information, and prepares the information in a presentable format to the end user.

8.1: THE FEED

An RSS feed is similar to a web page. The client system accesses an address on the Internet and is sent information over a connection. The information can be set up in one of several formats and is stored in different ways. The feeds are actually XML documents that are sent in response to a request sent to the web server. These documents can be hard coded, like a plain HTML file, or can be generated by a CGI script. XML documents are easy to read into a program and convert into objects for use in an languages like Perl, Ruby or Java. The schema used for the XML varies from feed to feed but generally fall into one of a few standardized formats.

Unfortunately, there is not a universally accepted format for RSS feeds. There are in fact several branches of the standard, many of which are not compatible with one another. To avoid these problems, we recommend the use of the Atom standard [6]. The Atom format is much nicer to

work with for many reasons. Atom is within an XML namespace and as of version 1.0, includes an XML schema. Furthermore, RSS is copyrighted and is frozen. Many RSS branches contain different formats for their payload, such as plain text and escaped HTML, but RSS does not indicate which is present. In generating an Atom feed, many parts are self-explanatory, however there are a few parts which may be confusing. The updated field contains a time stamp in a format that corresponds to RFC3339. Some examples:

```
<updated>2003-12-13T18:30:02Z</updated>
<updated>2003-12-13T18:30:02.25Z</updated>
<updated>2003-12-13T18:30:02+01:00</updated>
<updated>2003-12-13T18:30:02.25+01:00</updated>
```

The postfix to the time (followed by a '+') indicates the offset between the local timezone and GMT. A 'Z' indicates GMT. Another potentially confusing part of the specification is the id element of the entries. The ID field must be an IRI, as specified in RFC3987 [7]. The tag: format is a good way to achieve this. An example of this would be: <id>tag:mail-testing.cisco.com,2006-04-13:/patch?id=3941</id>. For more examples, see "How to Make a Good ID in Atom" at http://diveintomark.org/archives/2004/05/28/howto-atom-id. Finally, the link element with the attribute rel="alternate" allows the reader to provide a clickable link to the full text of the entry or feed.

8.2: THE READER

A reader is the RSS equivalent of a browser. It is given the address of an RSS feed which the user wants to 'subscribe' to. The reader then remembers that address and sends requests to it on a repeating interval. Unlike HTML, which has a predefined way of displaying each possible tag, the XML for each RSS standard is different. This results in very different ways of displaying information when using different RSS readers. There are a variety of options available on the different RSS readers. Most RSS readers give some sort of active notification to the user whenever it discovers a

new item on the feed. This notification can be in many forms, including playing a sound on the user's machine or displaying a dialogue box somewhere on the user's screen. Many RSS readers also have the ability to cache information received from the feed, so when an item is removed from the feed, the user can still read that item hours or even days later. There are many readers freely available on the Internet including some open source ones. One such reader is Feed 'N Read which is available at http://sourceforge.net/projects/fnr/. We took special note of this reader because it is free, open source, and written in Java. Being written in Java is most important, because we want Cisco employees to be able to use a standardized reader with similar settings across platforms, which is very easy to do in Java.

8.3: SUMMARY

Creating an RSS feed in our application took little time. Using Ruby on Rails to generate an XML document allowed us to integrate RSS into our program with little effort, due to the similar format of XML and HTML.

Once our application included an RSS feed the notification problem was solved. Cisco employees will be able to use the RSS reader they are most comfortable with to subscribe to our application's feed and receive notifications derived from their settings.

CHAPTER 9: CONCLUSION

We started this project with nearly no knowledge of Ruby on Rails or Cisco's notification system. We were asked to learn the existing system and a few problems with it. The first problem was that users were receiving too many e-mails to be able to determine which ones contained critical task assignments. The second problem was that users were required to constantly monitor their e-mail in order to receive their task assignments. If they were involved in a low priority task, they may miss a higher priority task assignment.

Our solution came in two parts. The first part is the mail filter tool. This tool has several components. It replaces the e-mail sending mechanism of Cisco's existing projects, like the Patch Manager, and stores those messages into a database. The mail filter tool also allows users to log in and configure e-mail delivery options. Users can specify whether they want to receive a message type immediately (normal e-mail delivery), never (the e-mail is blocked), or in a digest message. Digest messages are a compilation of all messages received over a given time period and are sent to the user in a single e-mail. Once messages are stored in the database a second script retrieves the messages, checks the recipients' delivery options, and delivers copies of that message as appropriate.

The second part of our solution is the notification tool. Users are given the ability to select certain messages in the mail filter tool that they want to be notified about. To implement the notification tool we created a Ruby script within the mail filter tool that pulls messages out of the database and uses them to generate an XML document. That XML document is called an RSS feed, which users can subscribe to with their favorite RSS reader. The RSS reader automatically pulls items from the RSS feed and gives the user the ability to customize how they are notified.

9.1: FUTURE WORK

The mail filter system has been deployed to a server accessible to all Cisco employees at the Boxborough campus. The system is temporarily running along side the current non-filtered e-mail system in order to make sure the mail filter system operates normally and can handle the daily load of e-mails deriving from the Patch Manager and Build Manager.

There are a few features we had planned to implement in the mail filter system. The current mail filter system automatically generates digest e-mails for all users at a system wide specified time. It would be possible to add a user-by-user setting for generating digest e-mails.

REFERENCES

[1] "Download Gmail Notifier". March 21, 2006. http://toolbar.google.com/gmail-helper/
[2] Seaspine Software. "TestTrack Pro". http://www.seapine.com/ttpro.html
[3] Hansson, David Heinmeiser. "Web Development that Doesn't Hurt". http://rubyonrails.org/
[4] "Technology at Harvard Law". http://blogs.law.harvard.edu/tech/rss
[5] Yeong, Howes and Kille. "X.500 Lightweight Directory Access Protocol". July1993. http://www.ietf.org/rfc/rfc1487.txt
[6] Nottingham, M. and Sayre, E. "The Atom Syndication Format". December 2005. http://www.ietf.org/rfc/rfc4287.txt
[7] Duerst, M. and Suignard, M. "Internationalized Resource Identifiers". January 2005. http://www.ietf.org/rfc/rfc3987.txt