Project Number: MQP-CEW-0703

Parsing and Analyzing Log Files

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

Christopher S. Henning

Christian A. Roy

Date: May 10, 2007

Professor Craig E. Wills, Major Advisor

Abstract

In many large-scale software and hardware systems, log files provide a crucial source of information to aid in debugging. However, when the system is complex these log files may be difficult to read. Often, entries are irrelevant, and combining and correlating different logs is difficult. In this project, we develop a tool to make a class of logs more useful to Cisco Systems. The tool filters log events based on specified information, as well as displays logs together for correlation.

Contents

List of Figures

Chapter 1

Introduction

Cisco Systems, Inc. provides a variety of networking and communication services, tools, and products that are widely used in the corporate world. These tools can differ widely in scope and purpose, but Cisco maintains support for them. Many times, providing this support can be expensive in both man-hours and money.

One of the products Cisco provides is a complex call-routing system to enterprise customers. This system enables companies to quickly and easily route calls to their correct destination, taking into account such factors as call center location, department, agent language, agent skill areas, and more. The system consists of several parts, including routers and various types of servers.

1.1 Problem

When something goes wrong, be it a call routed to the wrong place, a call taking too long to be taken care of, an agent not being logged in or out properly, or any number of problems, the main tool for finding the problem are the logs files generated by each piece of equipment. These files can be humongous and unwieldy, and are unfortunately not standardized with each other or even within a particular log. In addition, the devices can be configured to log with variable verbosity, so not all log files are generated equally. As such, it is often difficult to trace through the logs and find the problem. Cisco employees would usually start with the entire log file and remove irrelevant lines by hand, or sometimes start with a simple search for relevant lines (using grep, a common Unix text-searching tool, for example) and go from there. This approach, however, is time-consuming and can get confusing and hard to handle. In addition, when it is necessary to trace through multiple log files (as it often is), it becomes even more difficult to manage in a time-effective manner. The problem is that previously, no comprehensive tool has existed to make this job easier.

1.2 Requirements

Any solution we designed needed to meet several primary requirements in order to be of use to Cisco. Foremost, any software designed needed to be able to be run on Microsoft Windows. This was important because the most common operating system for Cisco employee's is Microsoft Windows. As such, it would be unusable to our target user base if it would not run on Windows. However, as some of Cisco's employees use alternate operating systems such as Unix or Mac OS X, cross-platform support was desirable if possible.

The second requirement was that it needed to use the existing log file formats. Since information in log files is generated at multiple points from many products, it was not feasible to propose a complete change of the format, though minor changes to specific event types were possible.

The third and final requirement was that it must be easily extended and modifiable. Although there are only four major log file formats currently, these formats may change in the future. Also likely, a new component or server type may be added that also needs to be included in the tool. As such, it is important that the tool be easy for others to change after we have finished the project and are no longer available.

1.3 Solution Approach

In approaching the problem our first task was to study the log files to try and understand the difficulties in parsing them. We began by watching a Cisco employee run through the hand parsing of some sample log files created. After we had a rough understanding of the log file, we decided to prototype a tool in Perl so as to test our knowledge. When this worked, we migrated to creating a solution in Java.

The primary reason we decided on this route was that while Java provides many advantages to long-term software projects, it can also be more difficult to accomplish certain tasks with it. By prototyping a tool, we could quickly see what assumptions we made were correct and which were not. Once we had that knowledge, it would then be an easier task to implement it in Java. In other words, we prototyped in a language that allows fast, flexible development so we could work out the conceptual problems before implementing a solution in a language that provides more extensibility to a constructed solution.

1.4 Road Map

This report provides more details about the background of the problem, what we used to tackle it, and what design decisions we made. In Chapter 2 we describe specific background knowledge necessary for the rest of the report.

In Chapter 3, we describe the problem we were trying to solve in more detail, and in Chapter 4 we describe our proposed solution, our actual solution, and the difference between them. In Chapter 5, we give a glimpse of our tool in action. Chapter 6 describes an alternate solution to the problem, the Call Flow Tool developed by a different branch of Cisco during the course of our project. Chapter 6 also compares our solution and the Call Flow Tool, including our recommendations on where to focus future work. In our final chapter, Chapter 7, we discuss final thoughts regarding the project. These thoughts include our accomplishments, future work to be done, and a concluding note on the project as a whole. Finally, we end by acknowledging all those who helped us with this project.

Chapter 2

Background

In this chapter, we discuss all the background knowledge required in the rest of the report. This includes information on regular expressions, an important tool used when parsing text, as well as the rationale behind which languages we used. Finally, it also includes the various tools we used while developing our solution.

2.1 Regular Expressions

One of the most crucial tools we use in parsing the log files is regular expressions. Regular expressions provide an extensive way to describe a pattern of text, including the ability to extract information from that text. One of the most popular forms of regular expressions are known as Perl Compatible regular expressions, as they are based off Perl's implementation.

A simple example of a Perl compatible regular expression would be 'bar', which would match the words 'bar' and 'foobar', since both contain bar. It is also possible to match the beginning and ending of a string via the characters `'^'` and `'$'` respectively. In this case, the regular expression `'^bar$'` would only match the word 'bar'.

Regular expressions can be further extended with character classes. In this case, one may specify a set of characters that are acceptable to match. To further our earlier example, the regular expression `'^[b|c]ar$'` would match both the words 'bar' and 'car'. Finally, it is possible to extract information from the captured string as well. In order to do this, you simply enclose the part of the expression to extract in parenthesis. In our example, the regular expression `'^([b|c])ar$'` would match the words 'bar' and 'car', capturing and extracting the character 'b' or 'c', respectively. Further information on regular expressions can be found at "http://www.regular-expressions.info"[1].

2.2 Language Choices

When developing a software solution, one of the most important choices one makes is the language to use. For our solution, we considered several languages. Of these languages, the primary candidates were C, Perl, Java, Python, and Ruby.

2.2.1 C

C is a systems programming language originally developed by Dennis Ritchie at Bell Labs between 1969 and 1973[3]. It provides low level access to the underlying hardware of a computer, giving access to memory via pointers. With an extensive library of functions, C and its derivative C++ are two of the most widely used languages on the planet.

2.2.2 Java

Java was created by Sun Microsystems in the early 1990s[2]. Borrowing heavily from C syntax, it is a wholly object-oriented language. A primary distinction between it and C/C++ is that it compiles down to byte code, which then runs in a virtual machine. As such, Java aims to be platform agnostic, with the motto "Write Once, Run Anywhere". It also comes with a full library of built-in class and functionally, including two frameworks for building GUIs, AWT and Swing.

2.2.3 Perl

Perl was created by Larry Wall and was first released in 1987[6]. It began primarily as a scripting language useful for manipulating text files, such as those commonly found in Unix configuration/administration. It was quickly adopted by many system administrators as the language of choice for these tasks, and eventually evolved into a type of swiss army knife of programming languages. It provides built in support for regular expressions, and a clean syntax for using them natively.

2.2.4 Python

Python was created by Guido van Rossum in 1991[7]. It aims to be a multi-paradigm, general purpose scripting language. Its primary goal is to promote program legibility, in that it embraces the concept that programs are written for people to understand first, computers to understand second. While it has support for regular expressions, there is not the same level of syntactic sugar for use as there is in Perl.

2.2.5 Ruby

Ruby is the most recently developed of the languages we considered. It was first released to the public in 1995, after several years development by Yukihiro Matsumoto[8]. It is a purely object-oriented language that also supports multiple programming paradigms. Unfortunately, its performance is the slowest of all languages considered. However, it has similar regular expression support as Perl, including the same clean syntax for using them.

2.2.6 Language Choice

Our conclusion was to use both Perl and Java for our project. Since C can have portability issues across platforms, we decided that it was probably not the best choice. Java, espousing the "Write Once, Run Anywhere" philosophy, neatly solves this problem and makes for a good option. However, most scripting languages (of which Perl, Python, and Ruby are all members of) also run cross-platform with little difficulty. The main advantage Java has over these languages is that it has a built-in library for GUI applications. Since our client is looking for a tool that will be easily usable and understood, a GUI is the ideal choice of interface.

However, scripting languages allow for more rapid development, something we definitely wanted to take advantage of given the time constraints of our project. To this end, we decided to use a scripting language to prototype our overall design and architecture. The options for this scripting language were Perl, Python, and Ruby. Since we had already determined that our final program would be done in Java, we knew we would be leveraging object-oriented programming, so Ruby and Python were both good choices for their clean object support. However, our experience with these was less than that of Perl and since the point was to rapidly prototype, we decided that implementing in Perl and then moving to Java and a more object oriented style would be faster than learning a new language. Additionally, since much of our work would be parsing text files, Perl's regular expression support was ideal, and while Ruby has similar syntax for regular expressions, it was the language we had the least familiarity with.

2.3 Tools

In any software project, a number of tools are used to make the task more manageable. These tools can range from simple editors to complicated Integrated Development Environments (IDEs). A brief summary of the tools we used follows.

2.3.1 Concurrent Versioning System

Every software project inevitably will have one or more (more likely a lot more) bugs in it as it is being developed. In addition, when development is done by several people, it can be difficult to coordinate who is working on what section, and even more difficult to merge changes made back into a central file that everyone has a copy of.

In order to solve these problems, the concept of version control came about. Concurrent Versioning System (CVS) is one implementation of a version control system[5]. It allows several developers to 'check out' a software project, make changes, and then 'commit' them back to the central repository. At this point, all other developers can 'update' their version of the code and CVS will automatically merge all changes. If two developers were working on the exact same section, when the second developer updates he will be notified of the conflicting changes and asked what to do about them.

2.3.2 Sourceforge

While CVS provides a way for teams to maintain code versions, it does not provide any way for teams to communicate beyond that. There is no concept of assigning tasks, releasing code builds, or just simple communication. To this end, Sourceforge exists[9]. Sourceforge is designed to make it easy for teams to develop and release software. It includes support for either CVS or Subversion (SVN) version control, as well as forums and a wiki.

2.3.3 Eclipse

Eclipse is an open source IDE originally created to ease Java development. It features syntax highlighting, auto-completion, a graphical debugger, and integrated CVS support[4]. It has a sophisticated plug-in system, which provide everything from support for other languages to GUI builders. While these plug-ins do provide support for other languages, Eclipse remains best at developing Java projects. As such, it was the primary tool used to develop the final version of our software solution.

2.3.4 Vim

Vim is a modal code editor designed for editing source code in a minimalist environment[10]. It can be run either at a command line or via a GUI, allowing people to edit in a variety of environments. It is a cross-platform editor, but remains most popular on Unix systems. While it has less features than Eclipse, notably no built-in debugging or CVS management, it has excellent syntax highlighting capabilities and was the primary tool used to develop the original Perl prototype of our solution.

2.4 Summary

To summarize, many decisions were made before we started serious development on our project. The first of these decisions was which language to use during development. We decided to prototype a solution in Perl to take advantage of its rapid development capabilities, as well as its strong and clean support for regular expressions. However, we also decided that our final tool should be implemented in Java for strong extensibility, and cross-platform and GUI support.

In addition to the decisions regarding language, we also chose several tools to help with regards to this project. These tools included version control systems, project management systems, and editors. Without these it would have been difficult to have developed our solution.

Chapter 3

The Problem

In this chapter, we discuss the problem we attempted to solve in more detail. It includes samples of the log files we worked with, as well as details on exactly why a tool is necessary to solve this problem.

3.1 Cisco's Log Files

For our project, we were mainly concerned with four log file types: Original Point Code (OPC) and Computer Telephony Integration (CTI), the two most important, and Protocol Independent Multicast (PIM) and JTAPI Gateway (JGW). The final part of the system is the Cisco CallManager (CCM), though we did not work with it at all. Both hardware and software products are a part of this system, and usually each part has a duplicate for redundancy. Each of these parts perform diverse actions as they work with each other to route calls, and each generates a log file detailing the actions it takes as its part of the process. Depending on the volume of calls and the particular setup, a certain piece of equipment may generate dozens of lines per second in its log. A diagram of the server configuration can be seen in Figure 3.1, however for our project the overall connection between components was less important than the log files themselves.

3.2 Sample Log Files

Figures 3.2 and 3.3 are samples of the log files we worked with. Figure 3.2 is the beginning of a CTI server log, and Figure 3.3 is the beginning of an OPC log. Both of them have lines that continue a ways off to the right, but for size and readability reasons the screen shots were cut off at 80 characters.

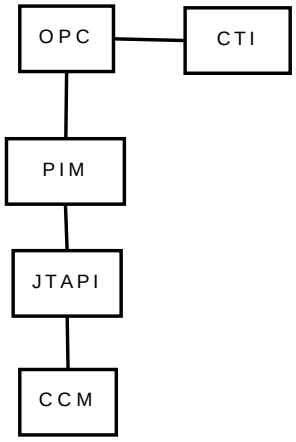

Figure 3.1: Cisco Server Setup

```
Events from March 13, 2007:
11:45:00 cg1A-ctisvr Trace: CTIApplication::ProcessTimeNotificationEvent - Time
11:45:16 cg1A-ctisvr Trace:

11:45:16 cg1A-ctisvr Trace: DEVICE_TARGET_PRE_CALL_IND - Instrument= RouterCall
11:45:16 cg1A-ctisvr Trace:                              ServiceSkillTargetID=-
11:45:16 cg1A-ctisvr Trace:                              AgentSkillTargetID=500
11:45:16 cg1A-ctisvr Trace:                              Var1= Var2= Var3= Var4
11:45:16 cg1A-ctisvr Trace:                              Var6= Var7= Var8= Var9
11:45:16 cg1A-ctisvr Trace:

11:45:16 cg1A-ctisvr Trace: AGENT_EVENT: ID=50005 Periph=5000 Ext=250005 Inst=2
11:45:16 cg1A-ctisvr Trace:               SkillGroupState=BUSY_OTHER SkillGroupD
11:45:16 cg1A-ctisvr Trace:               MRDID=1 NumTasks=1 MaxTaskLimit=1 Agen
11:45:16 cg1A-ctisvr Trace:               SkTgtID=5007 SkGrpNo=0x55e SkGrpID=500
11:45:16 cg1A-ctisvr SESSION 1: MsgType:AGENT_STATE_EVENT (MonitorID:0 Peripher
11:45:16 cg1A-ctisvr SESSION 1:         Simplified SkillGroupState:BUSY_OTHER S
11:45:16 cg1A-ctisvr SESSION 1:         SkillGroupID:5000 SkillGroupPriority:0
11:45:16 cg1A-ctisvr SESSION 1:         NumTasks:1 AgentMode:1 MaxTaskLimit:1 I
11:45:16 cg1A-ctisvr SESSION 1:         NumFltSkillGroups:0 ClientSignature:"CT
11:45:16 cg1A-ctisvr SESSION 1:         AgentInstrument:"250005" )
11:45:16 cg1A-ctisvr Trace:
```

Figure 3.2: CTI Server Log

3.3 Objective

As is the case with any product, problems with Cisco's systems crop up in regular use. A number of problems might occur; for example, a call might be incorrectly routed to an agent that does not speak the right language, or that is on a lunch break. Or an agent might not be logged in and out correctly, leading to payment problems. Or, a call might just take too long to be processed, delivered, and answered, meaning there is a decrease in the quality of service the company using Cisco's products provides to its customers.

As mentioned, Cisco's systems consist of a myriad of parts that work together and generate log files as they work. The files vary in verbosity and size, and there are a number of them. Each piece writes its own file in its own nonstandard way, too. And each device is highly configurable; any number of settings may be changed. But these files are where technical

```
Events from March 13, 2007:
11:45:00 pg1A-opc Trace: Peripheral::ProcessINRCMessageOut - 1 - !INFORM! - No D
11:45:16 pg1A-opc Trace: ProcessScriptResponse - !INFORM! - DID:4 on PID:4500 is
11:45:16 pg1A-opc Trace: Peripheral::ProcessINRCMessageOut - 8 - !INFORM! - Dial
11:45:16 pg1A-opc Trace: OPCCtrlBlock::ProcessPDDeviceTargetPreCallInd - !INFORM
11:45:16 pg1A-opc Trace: DEVICE_TARGET_PRE_CALL_IND (DATA PENDING) - PID=5000 Ro
                    NetworkTargetID=0 AgentInstrument= DelayQTime=0 SkillGroupDe
                    ServiceSkillTargetID=-1 SkillTargetType=-1 SkillTargetID=-1
                    SkillGroupSkillTargetID=5001 AgentSkillTargetID=5007
                    RtrCallKey=(148359-40305110) RtrSeq#=0 ECCSize=0
                    ANI=250007 Dialed#=211002 CED=
                    PV1= PV2= PV3= PV4= PV5=
                    PV6= PV7= PV8= PV9= PV10=
                    NICCallID={PCID=0 RCID=0 Remote=0,0 DlgID=0x0 RemDlgID=0x0 G
                    NIC_CalledPartyNumber=211002 CallTypeID=5006 InvokeID:19
11:45:16 pg1A-opc Trace: ProcessINRCMsgs - !INFORM! - Incoming INRC message is d
```

Figure 3.3: OPC Server Log

support personnel must go if they want to prevent problems such as the above from happening again. Due to the aforementioned reasons, finding the problem and why it occurred can be time-consuming and difficult. The support person must manually go through each log file, tracing calls and agents, and correlating events between different logs. They must also try to get a handle on the sheer amount of data the log files each present. Needless to say, working with the logs is often too much for a human to easily or quickly do.

To aid in finding problems, Cisco employees would often turn to `grep`, a tool for matching text lines to patterns. This approach may work, but even `grep` is limited at pulling out multi-line events, which most log types feature. Another tool Cisco employees would use is Notepad, or any other text editor. They would go through and manually remove any line that was irrelevant to the problem. These methods, of course, are time-consuming and prone to error.

So, the task of wading through the log files is monumental, and the tools to help this task are inadequate. Their problem is that solving some customers' problems is too hard to do quickly. Cisco envisioned our project as a way to make their lives easier. They desired a comprehensive tool, smarter than `grep` and quicker to use than Notepad.

3.4 Summary

Cisco's system includes different parts, all producing separate log files. These log files lack consistency amongst each other or event themselves. Currently, using these log files to troubleshoot problems is a time consuming process involving the manual filtration of irrelevant events. Cisco desires a comprehensive tool in order to reduce the time spent working with these log files.

Chapter 4

Our Solution

Once we had a strong understanding of the problem before us, we had a better idea of how to go about solving it. In this section, we discuss our original proposed solution, our actual solution and the reason they differ. In addition, we discuss the changes made to our tool due to feedback from Cisco employees. We also take an in-depth look at programmatically parsing these log files.

4.1 Our Proposal

Our original goal for accomplishing our task was to create a multi-phased project, with each phase having a separate deliverable. These phases were broken up into three primary parts, those parts being the act of parsing the log files separately, filtering the parsed events, and finally merging the results into some form of display that would highlight the source of the problems.

The primary rationale behind this approach was the limited time frame we had to work on this project. With only seven weeks to work, it was important that we break the problem down into sub-problems so that even if the whole project was unable to be completed, there would still be something to show for our work.

4.1.1 Parsing

In our proposed solution, parsing was the phase we scheduled the least amount of time for. The primary goal of this phase was to transform the different log files into a common format, that could then be easily filtered and merged. The primary idea for accomplishing this was to devise a new log file format to use as an intermediate step. In this case, we would transform the given log files to our new formats for easier merging and filtering.

4.1.2 Filtering

Filtering was the second primary phase of our proposed project. In this phase, we would take the new files created using our common format and filter them according to user input specifications. These specifications included filtering based on agent ID, call ID, or time frame.

4.1.3 Merging

Merging was the part of our project where we intended to spend the most time. In this phase, we would design a means to show the results of scanning several log files in a way that would highlight potential problems. Different means of doing this were considered, including time-based ladder diagrams, the showing of filtered output side by side, and also simply showing the output of all the parsed files in one list, sorted by time.

4.1.4 Summary

Our proposed solution was to have three core phases. These phases would be: parsing the files to a common format, filtering the files for relevant data, and then displaying this data to the user in some way. These phases were designed to be independent, so that if the project as a whole could not be completed there would still be a product of some use in alleviating the problem.

4.2 Actual Solution

While our proposal highlighted the three distinct parts of parsing log files, it took a somewhat idealized view of separation. When we actually began to implement our prototype, we noticed that their was a significant amount of overlap between the stages. In the end, our project became component based. Instead of separate phases, we had separate components in one cohesive application. We will begin by discussing the implementation of our prototype, followed by the design of our actual solution. While this may seem somewhat backward as normally design comes before implementation, one of the main reasons we prototyped was so that our actual result could influence our final design.

4.2.1 Prototyping

The first stage of code in our project was to prototype a solution in Perl. In this prototype, we would not care about efficiency or extensibility; the main purpose of it was to test that we could, in fact, parse the logs. However, since we were only prototyping, we did not want to spend the time required to parse all of the log file formats. For our prototype, we decided to focus specifically on CTI server logs. The reason we chose these logs was that they

had a clear, guaranteed line break between every event. This delimiter made it easier for us to tell when one event ended and another began.

Approach

Even though this was just a prototype, we did have a definite approach. Early on, we realised that it was difficult for two people to work on writing code to parse the exact same log file. Even though CVS would, for the most part, merge changes for us, the main problem we ran into was trying to avoid the duplication of work. Since we were, whereever possible, trying to write code that could parse more than one event we would constantly need to check to make sure we were not stepping on each others toes.

In order to prevent this problem, we decided to separate tasks. However, the question came of where to separate the tasks. Since our primary purpose at this stage was to be parsing in a log file and transforming it to a common format, there did not seem to be much overlap. It was at this point that we realised if we came up with a standardized internal format for logfile events, we would be able to work on both filtering, parsing, and even possibly merging concurrently.

Problems

After we decided to separate tasks, the only problem left was what internal structure to use for log file events. Since we were using Perl and dealing primarily with key value pairs, we decided to use Perl's built-in hashes. These hashes allow you to store a value to be referenced by a specific key, so it was an obvious choice given the problem. However, this did not quite solve all of our problems. Since the log file events were not consistent even across the same log file, certain values may not be associated with the exact same key across all entries. A perfect example of this in CTI logs is the extension number of the phone being called. In some events, it is referred to as 'device' and in others it is referred to as 'ext'.

Since filtering relies on being able to get common data from events, this was a major problem. In order to solve it, we decided to create a set of specific keys to index for these values. A simple example is that for any given event, its extension may be extracted via using the 'extension' key, regardless of whether it is also stored internally as 'ext' or 'device'.

An additional problem we had was the representation of time. For our project, knowing precisely when an event occurred was important. The log files can be configured to report each event's time to either the second or millisecond level, but we needed some way to store the time consistently for each object. Since speed was the primary point of our prototype, we did not want to spend too much time on devising a way to convert the times from any given input format to our own format, so we used what has become a somewhat standard way of representing time, we stored the number of

seconds since the 'epoch', midnight on January 1, 1970.

The problem with this approach is that it defined as the number of seconds since the epoch, but we need to support millisecond granularity. If we have that information, we can not simply discard it. However, all of Perl's built-in libraries expected seconds, so we could not modify that. In the end, we decided that the best course of action was to add the milliseconds as a fractional representation of the distance to the next second. While this resulted in some additional computational complexity regarding floating point numbers instead of integers, we decided it was a fair trade off considering we were not worried about efficiency at this stage.

After we had worked out these problems, work went quickly and we were able to complete our prototype in a timely manner.

4.2.2 Design

After our prototype was complete, we moved on to looking at how we could use what we learned for our final tool. In this case, we were also looking into how we could extend our Perl solution into one that takes advantage of some of Java's features, such as strong object orientation. However, we also had to compensate for the fact that Perl is a weakly typed language, meaning that variables can hold any type of data and it will try to figure out what type it is from the context. In Perl, using the string "2" in a mathematical setting would not result in an error, as it would attempt to convert it into a number for us.

Since the modifications to our proposal had worked out well with our Perl prototype, we decided to continue with a component-based model. In this model there would be a core framework that would use components to compute the result necessary for the other components. Fortunately, object-oriented programming is well equipped to handle this type of design. For our project, we developed four major parts. These parts are Events, Parsers, Filters, and User Interfaces. The merging portion of our original proposal was moved into the user interface, so as to allow different merging styles and specific interfaces to work with them.

Events

One of the core components of our design was the concept of an Event. An event represents a single action in a log file. In some log files, this corresponds to a single line, where in others a single event may span multiple lines. An event, much like in our prototype, must store certain specific data. It must store the exact string from the log file, as well as have separate specific fields for Call ID, Agent ID, and Extension, if the event is related to any of these three fields in any way. In addition, it must also store the time the event happened at, as well as provide a method that allows the Event to be printed

out as a string (possibly in a neater format than the string it was created from).

Parsers

Parsers are the components that, when given a log file, produce a list of Events. This is where most of the work is done, and also where most of the extendability comes into play. As long as there is a parser to take a log file and turn it into a list of events, then that log file is supported by our tool. This will be covered more in-depth in a later section.

Filters

Filters are what are responsible for taking a list of Events and trimming that list down to only relevent ones. They provide the 'core' component of our project, as the different User Interfaces interact with them directly (and no other component). For our project, the Filter was able to filter based on either Call ID or Agent ID, as well as on time.

User Interfaces

The final major component of our project is the User Interface. Since this is completely separated from the filtering and parsing of Events, it is possible to create generic user interfaces that work with any log file there is a parser for. For our tool, we designed two primary interfaces, one graphical and the other console based.

Summary

Due to a highly compartmentalized design, we were able to work on independent pieces of the project without worrying about interfering with each others work. In addition, this compartmentalized design makes it relatively easy to add support for new log file formats, since all that is needed is the crafting of a new Parser. It is likewise easy to build specialized User Interfaces for a given task, since they just interact directly with a Filter.

4.2.3 Implementation

As mentioned in the Design section, for our final implementation we went with a modular approach. In order to do this, we took advantage of several of Java's features, such as inheritance and implicit casting. Since our solution was designed to be modular, any individual piece could easily be replaced by another with similar functionality. In order to explain the pieces we created and the way they interact, we reference the UML diagram in Figure 4.1. We work from the bottom of the diagram towards the top, starting with LogEvent and moving through Parser, Filter, and finally the Interfaces.

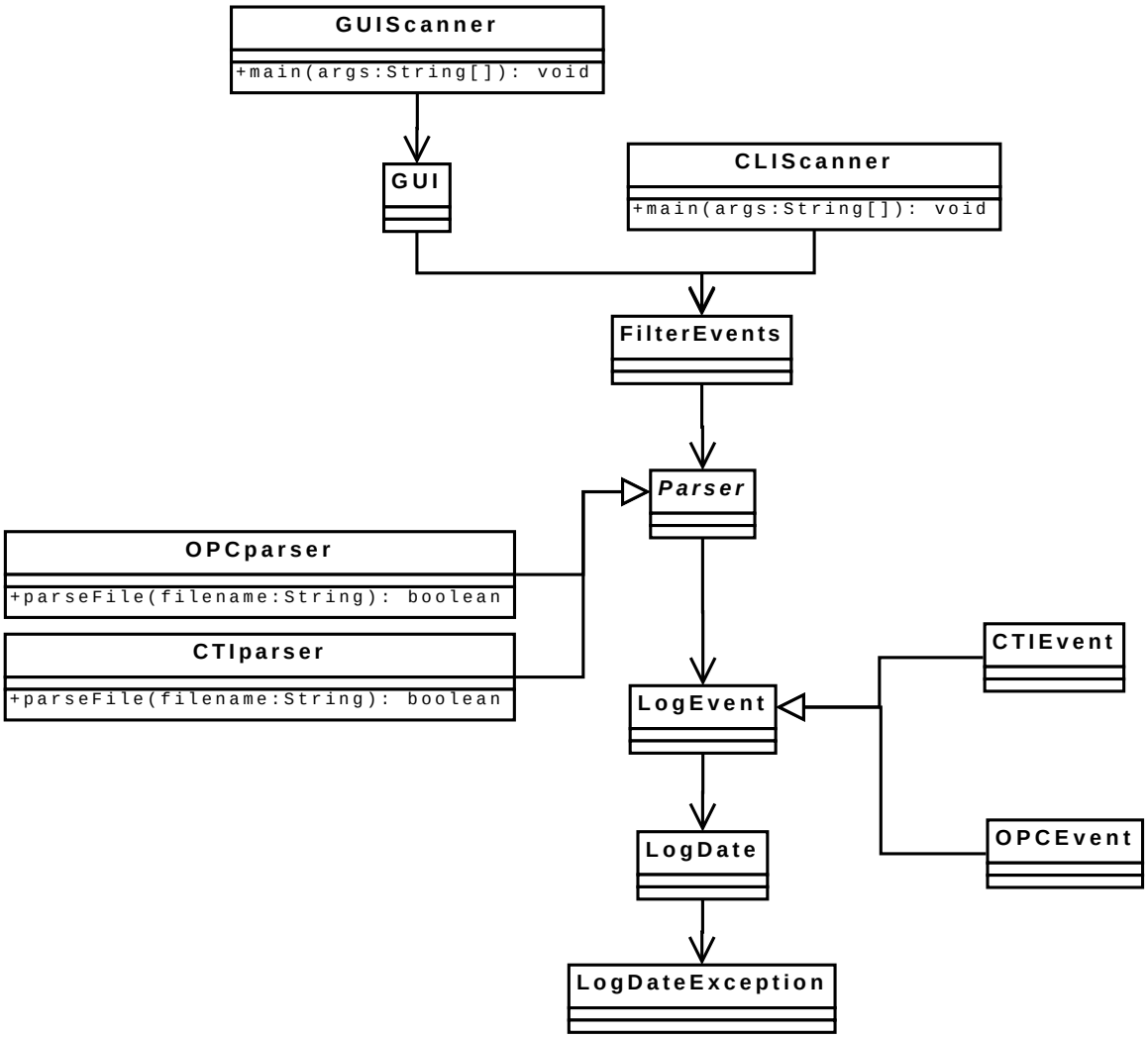

Figure 4.1: UML Class Diagram of Our Solution

Events

In our Design section, we spoke about the concept of an Event. For our actual implementation, we created a new superclass designed to store all the information we associate with an event, appropriately naming it LogEvent. This superclass contains fields for various data including name, agent ID, call ID, and extension number. It also includes a HashMap of all the 'key=value' pairs in the event. In addition, it includes data about the time the event occurred at. Since storing this data was such a problem in our prototype, we were careful to design our own consistent means of storing this data. To that end, we created the LogDate class.

The LogDate class is responsible for storing the time an event occurred at, as well as provide an easier way to print out that time in a consistent manner. In addition to this, it provided an easy way to parse a variety of timestamp formats into one format, via differing constructors. This, of course, leads to the possibility of being requested to create a LogDate from a non-supported timestamp format. To cover this eventuality, we created a LogDateException to be thrown if this was attempted.

This LogEvent was intended primarily as a base class to be extended off of for individual log file events. In our project, we extended this for two separate log files, the OPC and CTI logs. While the base class stores most of the data necessary for any event, subclassing provides an opportunity to modify the string representation of the event depending on the class, and this is primarily what we used it for.

Parsers

Moving upward from LogEvent in our diagram is the Parser. As in our Design section, the Parser is primarily concerned with taking a log file and transforming it into a list of LogEvents. Once again, the Parser class is intended as a base class to be extended off of. However, unlike our LogEvent, the Parser class is abstract. This means that if someone wants to use it, he must subclass it and implement any methods specified but not provided in the base class. In Parser, this primary method was called parseFile.

The Parser section is where the extensibility for new log file formats comes into play. In order to add support for a new format, all that is necessary is the creation of a new subclass to parse it. In this subclass, all that needs to be done is to create a new constructor and write a custom parseFile method. Unfortunately, this parseFile method can be complicated.

In order to write the parseFile method, an in-depth knowledge of the log file to be parsed is necessary. The parseFile method is responsible for transforming the given log file into a list of Events, and so this method must perform the difficult task of extracting the data from the different lines of the log file. In addition, it must take care to make sure the mandatory fields used for filtering are set in the LogEvent if at all possible. This is to simplify

the task of filtering events after the file has been parsed. For our project, we created two Parsers, one for OPC logs and the other for CTI logs. Our experiences with parsing these logs can be found in the later section "In-depth Parsing".

Filtering

Continuing with our diagram, the next component is the Filter. As mentioned in Design, once data has been parsed out of a log file, it is up to the filter to decide what to keep and what not to keep. Given a list of all events, along with the type and value of an identification number and a range of time, the filter goes through and constructs a list of only the relevant events. If the filter is given a filename instead of a list of LogEvents, it can create the appropriate parser in order to create the list and continue filtering from there.

In order to filter the list of LogEvents once it has them, it uses the special fields defined in the LogEvent classed, namely the Agent ID, Call ID, Extension Number, and timestamp. This is, at first glance, a relatively simple task. It merely has to go through the list and discard any event not in the time range, or that does not contain the specific value we are looking for.

Unfortunately, it frequently is not that simple. Often times, events are related but not directly. As an example, in the CTI server logs, an event rarely if ever contains both an Agent ID and a Call ID, so getting information on all calls taken by a specific Agent is difficult. Fortunately, they are related by extension number, but this is not enough information to use in a single pass system. In order to allow these kinds of difficult relationships, we allow the user to set what we called the search depth.

The search depth is basically how many passes through the list the tool will go trying to relate events. For example, on the first pass through, we may learn that the given Agent took calls on a specific Extension. If we only do this one pass before filtering, we can then keep all events that have either the Agent ID or the Extension number. If we do two passes through, we may learn about a Call ID associated with that Extension number, and we can then keep those events as well.

It is important to note that we do these passes before any other filtering is done. This does result in a performance sacrifice, especially with large log files, but we determined it was better to make this sacrifice then to possibly miss out on a connection from an event that would have been filtered.

User Interfaces

The final section of our diagram is dedicated to the user interfaces. These interfaces are responsible for taking input from the user, giving this information to the filter, and displaying the resulting list of LogEvents in an intelligent manner.

In our project two interfaces were created, with three modes total. Each mode was created for a different purpose. The first interface created was the command line interface. This was designed to run in a console or terminal, and provide output directly back to the screen. This interface has two modes, the batch mode and the interactive mode. The primary difference between these modes is that batch mode was designed for a single scan, where interactive mode was designed to be more of an exploratory environment. In batch mode, you set parameters as command line arguments, the tool runs, and then results are displayed. However, in interactive mode, you are given a prompt where you can set parameters. After initiating a scan, the results are displayed and you go back to the prompt, where you can change parameters until you get the output desired.

The second interface created was an interactive graphical user interface (GUI). It offers the same functionality as the interactive command-line mode, allowing the user to input specific parameters and see the output of the scan. However, it also adds additional functionality, such as allowing the user to scan more than one file at a time. In addition, it allows the user to directly remove events from the list displayed, as well as save the list to a file.

4.2.4 Summary

Our actual solution differed greatly from our original proposal. While the separation of the project into three parts was maintained, the phasing of implementation was not. Due to insight gained during prototyping, we were able to greatly simplify our design and implementation for the final result, creating a modular and extensible tool.

4.3 In-depth Parsing

While we discussed Parsing earlier in our Actual Solution section, that was mainly intended as an overview of the parser component, not our experience with actually parsing the log files. A more in depth look at the parsing of the two logs were worked with is contained in this section for anyone wishing to know what pitfalls to look out for.

4.3.1 Parsing CTI logs

Our primary means of parsing CTI logs was through regular expressions. Thankfully, in CTI logs there are a few consistencies that make parsing this log easier than the OPC logs we worked with. Determining where an event starts, its name, and its timestamp can be obtained with one regular expression. After that, determining what subsequent lines belong to the same event is also easy. Once this knowledge is obtained, parsing out the “key=value”

pairs can be done with a separate (and complicated) regular expression. Some of the key points we came across when parsing CTI logs are as follows:

- The date is on lines of the form: `Events from March 13, 2007:`
- All lines start with a timestamp (which can have ms), the server name, and the server type, e.g. `11:45:00 cg1A-ctisvr`
- Events may span multiple lines; different events are separated by a blank line
- After the timestamp and server info, there will be either `Trace:` or `SESSION X:` where X is a number. Normally, we do not care about session-level messages.
- On a line that begins an event, after the time, server info, and `Trace:` is the event name. Sometimes after the name is `-` , sometimes `:` , and sometimes there is a value in parentheses you might care about, e.g in. `11:46:33 cg1A-ctisvr Trace: CSTA_PRIVATE (RTP_STARTED_IND)`
- Subsequent event lines have a group of spaces after the `Trace:`, then any number of `key=value` pairs.
  - Keys never have spaces in their names.
  - Values may be blank
  - Some values have spaces in them! e.g.:

    `RouterCallKey=148359 40305115`
- Calls and agents are related by extensions. No event has both a call ID and agent ID in it, but many have an agent and an extension, or a call and an extension.
- Most values that are call IDs are of the form `id.ext(s)` e.g.:

  `callID=22541571.250007(s)`

4.3.2 Parsing OPC logs

Parsing OPC server logs is more complicated than parsing CTI logs because there are more inconsistencies. Determining where an event starts and what subsequent lines belong to it, along with getting the event name, can still be done with a (complicated) regular expression, however parsing the "key=value" pairs is made more difficult by the fact that the values may contain spaces or be enclosed in parentheses. We found that, while it was likely still possible to build a regular expression to do this, it was simpler to separate the strings based on spaces, then reconstruct information based on those tokens. As with CTI logs, the following are some of the key points we discovered in parsing these log files.

- The date (which may change) is on lines like:

  `Events from March 13, 2007:`

- Events may span multiple lines. Most of the time, only the first line contains a timestamp, however some events' inner lines have the normal header (next three items), then more than one space, and then more `key=value` pairs.
- Time stamps are of the form `11:45:16`; milliseconds are possible
- After the timestamp is the server name and the type: `11:45:16 pg1A-opc`
- After the server info is "`Trace:`" - `11:45:16 pg1A-opc Trace:`
- After `Trace:` is the event name, e.g.:

  `DEVICE_TARGET_PRE_CALL_IND (DATA PENDING)`

  - You may care about the values in parentheses
  - Event names do not contain spaces, but the string in parentheses might

- Most event names are delineated from the rest of the info by a `-` , but some are by `::`
- After the marker in the previous item, `key=value` pairs ensue.
  - Keys may not contain spaces
  - Values may be null
  - Values may contain spaces! e.g.: `SkillGroupNum=1374 (0x55E)`
  - Some values are in parentheses, but there are additional `key=value` pairs inside them, e.g.:

    `clearedCall=(CallID=4 Device= Type=Static)`

- Call IDs and agent IDs are directly related in at least one event.

4.4 Feedback

As we worked on our project, we sent progress updates to our supervisor at Cisco, Rajendra Joshi, approximately once a week. When we started having working builds, we included those as well, requesting feedback regarding and changes or problems with our tool. The feedback was mostly positive, with some minor requests to include support for different events in a specified log

file. In addition, we also received feedback regarding new features to add, as well as a few bugs to fix.

One of the primary forms of feedback we received was in the form of a new log file to work with. Up until that point, we had been working with sample log files provided by our supervisor, but shortly after we released our first major build we were informed that it did not work on this specific log file. Upon further research we discovered that the problem was caused for two reasons. The primary reason was that a specific event type had a different format than the one in our sample logs. After fixing this, however, we ran into a larger problem which was that the Java Virtual Machine(JVM) was running out of memory. In order to fix this problem, we had to pass specific arguments to the JVM before running our tool. To simplify this process for the end user, we created a batch file that can be used to run our tool with appropriate settings.

The other primary feedback we received was in requests for functionality. At the request of one of our contacts at Cisco, Bill Lapp, we implemented a means to save the output of the GUI interface, as well as the ability to remove events from the list created by the GUI before saving.

4.5 Problems with Our Solution

As our project only had a limited time frame, our solution is not as complete as it ideally would be. We did not have time to add certain features or improve on certain problems, but what it does, it does well (if slowly). Due to the size of the logs files and the amount of information we store about them, the memory requirements are high. It takes approximately twice the amount of memory to store a log file than it takes up on disk. In order to use this amount of memory in the JVM, we created batch files that pass arguments in order to tell it to allow us to use more memory. The memory footprint could be reduced though, for example by not storing the raw parsed line once the parser is done extracting information. Filtering events is also somewhat slow, because of the necessity of passing over the list of parsed events multiple times. This can be reduced by decreasing the search depth. Past performance problems, there are some features that were planned that were not implemented. One is the ability to cancel a scan in the middle of it; currently this is not possible due to our design, but it would be nice to modify our program to enable this. Another is the graphical display of a call flow, for example in a ladder diagram. This was a desired feature, but unfortunately due to time constraints we were unable to accomplish this.

4.6 Summary

In order to solve the problem set before us, we went through several important steps. First, we created a proposal of how we intended to solve the problem.

However, shortly into prototyping we realized that this proposal would not work, and so modified our project structure. After creating a quick prototype in Perl, we looked at the lessons we had learned from this and designed a final solution in Java. Through feedback from employees at Cisco, we refined our tool to better suit their needs. Unfortunately, there are still some problems with our solution due to time constraints, the primary problem being that it does not support ladder diagrams as was originally desired.

Chapter 5

System in Action

For this demonstration of the system in action, we will be using the example CTI server log provided to us. All three user interface modes will be demonstrated. We will first look for events associated with agent 50005, and then with call number 22541572.

5.1 Command-Line Interface

5.1.1 Batch Mode

The first interface is the command-line batch mode, in which arguments are given on the command-line invocation of our project. An example of this mode (slightly edited for length) can be seen in Figure 5.1. There, the file "cliscanner.jar" is being run, and given the arguments "-a 50005" for agent number 50005, and "ctisvr.txt" to give it the example log file. Though only the beginning of the output is pictured, it can be seen that the agent has been associated with call number 225415721, and so events related to that call are also being displayed. If we wanted to, we could specify other parameters on the command line, such as the start and end times to look between.

```
>cliscanner.bat -a 50005 ctisvr.txt
03/13/2007 11:45:16:000: AGENT_EVENT: AgentAvailabilityStatus=0
SkillGroupState=BUSY_OTHER SkTgtID=5007 Sig=CTIOSServer
MRDID=1 AgentMode=1 NumTasks=1 ClientStatus=0x1 ICMAgentID=5007
CurLine=-1 Ext=250005 SkGrpID=5000 ID=50005 OverallState=RESERVED
SkGrpNo=0x55e Inst=250005 OverallDuration=0 Reason=0
03/13/2007 11:45:16:000: CALL_RECLASSIFIED: newSource=22541571.250007(s)
oldCallID=4.(s) Periph=5000 newDest=
(etc)
```

Figure 5.1: Command-Line Batch Mode

5.1.2 Interactive Mode

The second available interface is command-line interactive mode. Here, the user can specify parameters in multiple steps, and also scan multiple times in one session, perhaps changing values in between scans. An example of interactive mode can be found in Figure 5.2 (slightly edited for length). Here, a time frame has been specified, and the program has only displayed related events in that time frame. Searching for call number 22541572 this time, we can see that it is also associated with agent 50005, and so events related to that agent are being displayed. Since the agent took a number of calls in the example log, but we only wanted to see about call 22541572, specifying a time frame was one way to narrow down the search. Another way would be to reduce the search depth so that only events that specifically contain that call ID are displayed.

```
Interactive mode!
>call 22541572
>file ctisvr.txt
>start 03/13/2007 11:45:54
>end 03/13/2007 11:46:03
>scan
Scan requested!
03/13/2007 11:45:54:000: CALL_CREATED: Wrapup= Type=1(ACD_IN)
Var1= ECCsize=0 Var2= Var10= Dest= Disposition=0(INVALID)
CallID=22541572.250007(s) Periph=5000 UserToUser=
03/13/2007 11:45:54:000: AGENT_EVENT: AgentAvailabilityStatus=0
SkillGroupState=BUSY_OTHER SkTgtID=5007 Sig=CTIOSServer MRDID=1
AgentMode=1 NumTasks=1 ClientStatus=0x1 ICMAgentID=5007
CurLine=-1 Ext=250005 SkGrpID=5000 ID=50005 OverallState=RESERVED
(etc)
```

Figure 5.2: Command-Line Interactive Mode

5.2 Graphical User Interface

The final interface developed was the graphical user interface. Using the power of Java's Swing library, we created an interactive and graphical method of using our program. This method is far more powerful than command-line mode because it allows the parsing and merging of multiple log files, removing unwanted events from the output text box, and saving of the output to an external file. It also allows multiple scans and the changing of parameters in between them, like the interactive CLI mode. This mode is pictured in Figure 5.3. Here, note that the search depth has been set to 0, meaning that

only events that specifically mention the desired call ID are displayed, and not any events related to any associated agents or phone/device extensions.

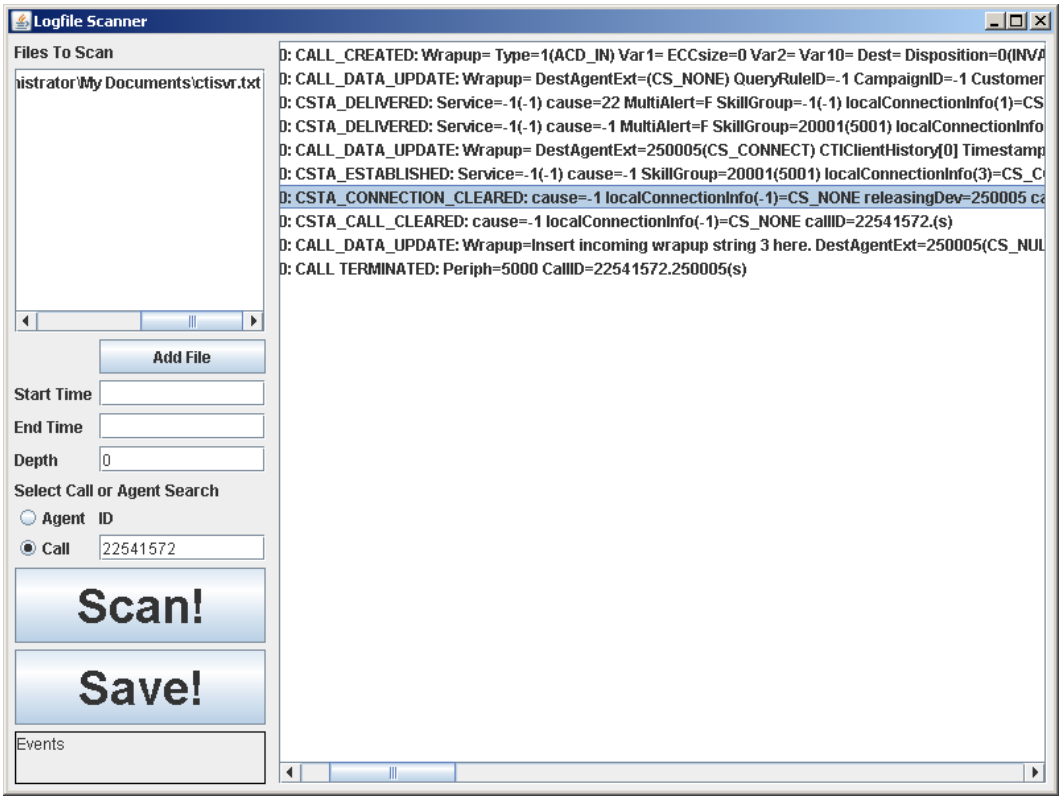

Figure 5.3: Screenshot of our GUI

5.3 Summary

We created three different UI modes over the course of the project. Each offers a different way to parse and view log files. Command-line mode offers a batch and an interactive mode, and the graphical user interface offers a more powerful and user-friendly mode.

Chapter 6

Call Flow Tool

Towards the end of our project at Cisco systems, we were informed about a tool recently developed in-house at another department. While development on the tool had been going on for some time, word had not yet spread about it.

The tool, developed by Jim Brikman, was designed to help the Customer Voice Portal team in much the same way our tool was supposed to help the team we were working with. The primary differences between our tool and his are the user interface and the means of adding support for new log files.

6.1 User Interface

The Call Flow Tool as developed currently has support for timing diagrams via ladders. It also has a dynamic means of filtering events based on fields they contain, as well as the ability to parse several log files at once. As can be seen in Figure 6.1, the ladder diagram is dynamically and generically generated based on information parsed from the file.

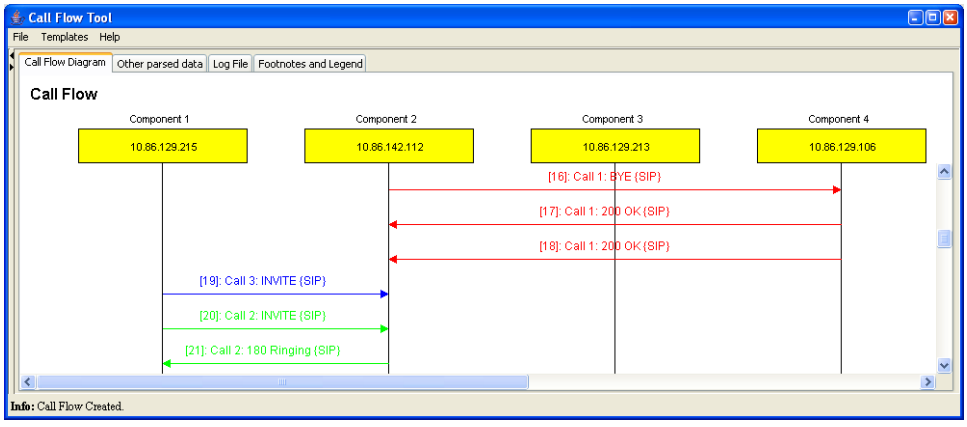

Figure 6.1: Call Flow User Interface

6.2 Extensibility

The Call Flow Tool provides support for future changes and log file formats via XML templates. These templates describe the format of the log file, as well as specifying the data to extract. Figure 6.2 shows a high level view of this XML, while Figure 6.3 shows a closer look at some of the work that goes into extending it.

```
– <MessageType name="CVP40_SIP">
    <TimestampFormat>MMM dd yyyy HH:mm:ss.SSS Z</TimestampFormat>
  + <RequiredFields></RequiredFields>
  + <OptionalFields></OptionalFields>
  + <DefineLines></DefineLines>
  + <FirstLines></FirstLines>
  + <AdditionalLines></AdditionalLines>
  + <EndLines></EndLines>
  + <LineAction name="add_text" for="TCPReceive, TCPSend, UDPSend, UDPReceive"></LineAction>
  + <LineAction name="set_received1" for="TCPReceive, UDPReceive"></LineAction>
  + <LineAction name="set_received2" for="TCPSend, UDPSend"></LineAction>
  + <LineAction name="set_response_code" for="Response"></LineAction>
    <MessageProcessingClass>com.cisco.cvp.CallFlow.SIPMessageProcessor</MessageProcessingClass>
  + <Automata name="sipFSM" groupBy="GUID"></Automata>
  </MessageType>
```

Figure 6.2: Overview of the XML format

```
+ <DefineLines></DefineLines>
– <FirstLines>
  – <Line name="TCPReceive" ignoreCase="true">
      <Literal>Received message on</Literal>
    – <PXML>
        {% $starterTrace %} {% $Literal %} {% PROTOCOL %} local[[ port = {% IGNORE %} ({% SENDER %})]] remote[[ port
        = {% IGNORE %} ({% RECEIVER %}) ]], {% IGNORE %}
      </PXML>
    </Line>
  – <Line name="TCPSend" ignoreCase="true">
      <Literal>Sending message on</Literal>
    – <PXML>
        {% $starterTrace %} {% $Literal %} {% PROTOCOL %} local[[ port = {% IGNORE %} ({% SENDER %})]] remote[[ port
        = {% IGNORE %} ({% RECEIVER %}) ]], {% IGNORE %}
      </PXML>
    </Line>
  – <Line name="UDPSend" ignoreCase="true">
      <Literal>packet on</Literal>
    – <PXML>
        {% $starterTrace %} Sending {% PROTOCOL %} {% $Literal %} {% SENDER %}:{% IGNORE %}, destination {%
        RECEIVER %}:{% IGNORE %}
      </PXML>
    </Line>
  – <Line name="UDPReceive" ignoreCase="true">
      <Literal>packet on</Literal>
    – <PXML>
        {% $starterTrace %} Received {% PROTOCOL %} {% $Literal %} {% RECEIVER %}:{% IGNORE %},source {%
        SENDER %}:{% IGNORE %}
      </PXML>
    </Line>
  </FirstLines>
+ <AdditionalLines></AdditionalLines>
```

Figure 6.3: In-depth look at the XML format

In addition to support for new log files via XML templates, the Call Flow Tool also provides support for SCXML automata for analysis. This allows the tool to be extended to support analysis for many types of log files.

6.3 Comparison

The obvious question at this point is whether the Call Flow Tool can be used to parse the log files we have been working with, and if so how much work it would take to do so. As mentioned earlier, the Call Flow Tool provides a means of extending support to new log files via an XML template file.

This template format is a powerful and versatile means of describing the individual log files. It allows the user to specify exactly what each line of a log file may look like, including an option to extract specific data for later use. However, while this is a powerful means of describing data, it does run into some problems when adapting it for use with the log files we worked with.

The primary difficulty is that the log files we worked with are, as mentioned earlier, non-standard. The fact that they lack standardization even across similar entry types adds new difficulties in using a tool that captures based off explicit declarations. This is not to say that it is impossible, merely that it will be a time-consuming and difficult process.

In spite of these difficulties, the Call Flow Tool does have several advantages over our tool. The primary advantage to the Call Flow Tool is that the developer is a long-term employee, as opposed to one working only on this project. That means that Cisco will have continued access to the developer for support even after we are gone.

Another major advantage is that this tool is already designed for producing ladder diagrams representing time flow. As this was the goal for our project, but something we were unable to complete due to time constraints, this is a major reason to look into adapting this tool instead of continuing to extend our own.

6.4 Summary

In conclusion, the Call Flow Tool provides a powerful means of parsing log files. While there may be some problems adapting it to include support for the log files we were working with, primarily due to inconsistencies in the log files themselves, the benefits outweigh the disadvantages of doing so.

Chapter 7

Final Thoughts

7.1 Accomplishments

Over the course of the project, many of our goals were accomplished. First, we produced a working prototype of a CTI log scanner in Perl. This prototype laid out the framework that we would later use in Java, and let us run into design and coding issues, allowing our subsequent Java version to be better-planned. It also let us run into and solve implementation issues, making future work easier. This Perl deliverable was a fully-functional program that offered both an interactive and a batch mode interface for scanning CTI server log files.

Once we were done with the Perl prototype, we moved to Java. Using what we had learned previously, we were able to start with a better and smarter design which also used the power of object-oriented Java. In Java, it was much easier to create a more powerful framework of objects and classes that would allow easy extension. We first produced a clone of the CTI log scanner, using what we learned and produced in the Perl version to speed things along. We then added an OPC log scanner, which was relatively easier because our design minimized the actual amount of code that was necessary. At the same time, we also produced a GUI version of the interface, which provided another way to interact with the program. Once that was done, we started to focus on improving the output of the program, making it more usable based on feedback received from Cisco Employees.

Shortly after this point, we were informed about the existence of the Call Flow Tool, and were asked to investigate whether it was a viable solution. We did some research on this, including exchanging e-mails with the developer, and concluded that it should be a viable, if time consuming to implement, solution to the problem.

7.2 Future Work

At this point, future work regarding our project mainly takes the form of creating differing ways to display the data. In addition to this, parsers do still need to be created for other log files, particularly for the JGW and PIM components. There are also a few interface options requested during our presentation of the project to Cisco, however for the most part interest seems to be in moving from our work to extending the Call Flow Tool to meet their needs.

7.3 Conclusion

The tool we created should help in analyzing and troubleshooting the two log files we worked with. While we did not have time to create any of the more complicated display options such as a ladder diagram, the creation of a tool that will automatically search through a series of log files and display only the relevant entries should provide a substantial boost to productivity.

However, while our tool does serve to reduce the problem, it makes more sense for Cisco to continue extending and working with the Call Flow Tool developed in another department. The call flow tool provides an extensible means of parsing almost any log file via an XML description. In this report, we have attempted to describe in detail some of the pitfalls that may be run into while attempting to parse the log files we worked with for precisely this purpose.

7.4 Acknowledgements

We would like to acknowledge the following people who helped us in completing this project. They are list in no particular order.

- Professor Craig Wills, for his assistance and guidance throughout the entirety of this project,
- Professor Gary Pollice, for his assistance in setting up our Sourceforge project and CVS repository,
- Rajendra Joshi, for managing our project,
- Bill Lapp Jr., for his many contributions and suggestions, as well as his help in testing our software tool,
- Ellen Garvey, for approving our project, and
- Andrew Socha, for providing a desktop to develop and test on.

Bibliography

[1] Goyvaerts, Jan. "The Premier Web Site about Regular Expressions" http://wwww.regular-expressions.info

[2] Sun Microsystems. "The Source for Java Developers". http://java.sun.com

[3] "Your resource for C and C++". http://www.cprogramming.com

[4] "The Eclipse Foundation". http://www.eclipse.org

[5] Price, Derek. "An Introduction to CVS". http://www.nongnu.org/cvs

[6] "The Perl Directory". http://www.perl.org

[7] "Python Programming Language -- Official Website". http://www.python.org

[8] "The Ruby Programming Language". http://www.ruby-lang.org

[9] "Sourceforge.net". http://www.sourceforge.net

[10] "Vim the Editor". http://www.vim.org