A Log Mining Approach to Failure Analysis of Enterprise Telephony Systems

Chinghway Lim[1,*] *Navjot Singh*[2] *Shalini Yajnik*[2]
[1] Department of Statistics, University of California, Berkeley, CA 94720
[2]Avaya Labs, 233 Mt. Airy Rd., Basking Ridge, NJ 07920
lim@stat.berkeley.edu, {singh,shalini}@avaya.com

Abstract— **Log monitoring techniques to characterize system and user behavior have gained significant popularity. Some common applications of study of systems logs are syslog mining to detect and predict system failure behavior, web log mining to characterize web usage patterns, and error/debug log analysis for detecting anomalies. In this paper, we discuss our experiences with applying log mining techniques to characterize the behavior of large enterprise telephony systems. We aim to detect, and in some cases, predict system anomalies. We describe the problems encountered in the study of such logs and propose some solutions. The key differentiator of our solutions is the use of individual message frequencies to characterize system behavior and the ability to incorporate domain-specific knowledge through user feedback. The techniques that we propose are general enough to be applicable to other systems logs and can easily be packaged into automated tools for log analysis.**

I. Introduction

Large complex telephony applications, like enterprise Voice-over-IP (VoIP) systems, call-centers, and contact centers, require high reliability and availability in order to prevent any downtime that would lead to loss of business, loss of revenue and in some cases, loss of life. Most of these systems are already designed with very high availability in mind. However, as with any complex software system, bugs, misconfiguration, and other operator problems, do cause occasional failures and bring the system down. When a downtime occurs, as part of recovery procedure the system administrators and the service support staff may look at the system trace and debug logs to identify the cause of the failure. The current process of exploring the log file is fairly manual in nature. A large software system may consist of a number of modules operating in a multi-threaded environment, with each module writing its debug and trace messages to a log file. We believe that the system logs (both debug and trace logs) depict the state of the system fairly accurately. They may allow us to detect and sometimes predict, the occurrence of anomalous behavior and take timely action to prevent future downtime.

Study of system logs to characterize system and user behavior has been a focus of research for some time. Web logs containing user search patterns and navigation behavior [1] have been studied to improve web site usage and also to provide users with targeted product advertising. Syslogs, transaction logs and error logs have also been mined for detecting failures and/or anomalous behavior in a system [2], [3], [4]. Log visualization [5] has been another area of fertile research, where abstract visualization of log file has been used as a tool for determining system state.

Early work on the study of logs focused on statistical modeling of errors and failure prediction based on the models. Iyer et al [6] noted that the alerts in the DEC VAX-cluster systems were correlated and based on this observation developed models for behavior of these systems. Work by Nassar and Andrews [7] showed that there was an increase in the rate of non-fatal errors before the system went into a failure mode. Data mining approaches like frequent pattern mining and sequential pattern mining have also been used to detect commonly occurring event patterns before a failure [8], [9], [10]. The paper by Vaarandi [10] clusters event logs based on the message content and then detects anomalies by finding deviations from these clusters. The work by Oliner and Stearley [11] describes the problems encountered in studying large logs generated by supercomputer systems.

Despite the fact that the system *trace and debug logs* contain a wealth of state information, they have been an under-utilized resource. This may be due to the highly unstructured nature of the contents of such logs. Most of the previous work on log analysis is based on searching for and mapping a set of predefined textual patterns in the log. In contrast, trace/debug logs are usually generated by system developers for testing and in-field debugging and hence may not have any pre-defined textual patterns that can be monitored. Indeed, the problem there is really to detect patterns which depict aspects of system behavior, irrespective of the textual content of the individual log messages.

The main focus of this paper is to use a combination of data mining and statistical analysis techniques on the logs obtained from large enterprise telephony systems and show how such techniques are very useful in detecting and in some cases predicting failures and anomalies in the system. Currently, system administrators and service support personnel rely on experience and expert information to extract useful information out of the log files. As an example, a service support personnel at a customer site may manually browse through the log file to detect the cause of a system crash. Our eventual goal is to quantify and automate most of these skills to make evaluation and predictions based on the log files more efficient. The techniques that we present in this paper form the basis for an automated tool that we are currently in the process

*This work was done while the author was at Avaya Labs.

of building.

The rest of the paper is organized as follows. Section II gives an overview of the enterprise telephony communication application that we studied. Section III explains the goals of the analysis. Section IV discusses the techniques that were used to reduce the data through preprocessing. Sections V and VI, respectively, apply data mining and statistical analysis techniques to logs with known failure markers and logs with no failure markers. Section VII concludes with a discussion of the results.

II. Enterprise Telephony System

Avaya is a vendor of enterprise voice communication equipment and applications. An enterprise telephony system (Figure 1) consists of several components which communicate with each other to provide the telephony service to end-users. These include the Communication Manager (CM), the gateways, and a number of IP-capable end-points. The CM implements the bulk of the telephony features. It performs various tasks such as phone authentication and registration, call routing, call signaling, and, call initiation/termination. In a typical large deployment, a single CM is capable of supporting hundreds of thousands of IP phones with loads of similar scale in terms of peak busy hour calls.

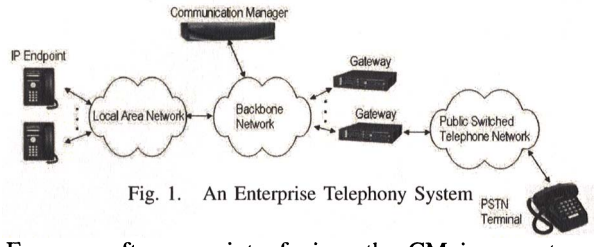

Fig. 1. An Enterprise Telephony System

From a software point of view, the CM is an extremely large and complex piece of software with millions of lines of code that have evolved over a decade of development. During operation, the CM comprises of more than 50 processes that handle the computing and communication tasks described above. Each process writes trace and debug messages into a shared CM log file. Below is an example of the log generated by two processes - *proc1* and *proc2*. The log follows the pattern of *Date:Time:Sequence:ProcessName (PID): Priority :[Payload]*. The *Payload* part of the log consists of trace and debug messages generated by each process.

```
20060118:032918946:2445:proc1(2548):HIGH:[timRestVar: time thread to sleep]
20060118:033026975:2446:proc2(2249):HIGH:[IntchgReqFail: errcode=1]
20060118:033116820:2451:proc2(2249):MED:[Being told to Go Active!!!]
20060118:033116820:2454:proc2(2249):MED:[standby→active :interchange..]
20060118:033116820:2455:proc2(2249):MED:[State Transition:standby to active]
```

As one would expect from a debug/trace log, the payload part of each line does not follow any strict pattern and can be classified as *unstructured*.

A. Log Collection

As mentioned in Section I, the motivation for this work was to understand the failure behavior of CM deployments in the field. So, the first order of business was to collect the debug/trace logs from live systems. Due to time and bandwidth constraints imposed by the customer environment, the log size was restricted to about *four hours* during failure of which pre-failure log was approximately three hours. We retrieved 714 log files from 460 live CMs that spanned 23 different releases. Each of these log files had a failure that caused a system restart. We applied our techniques to all 714 logs to study their behavior. The corresponding analysis and its results are described in Section V. Additionally, we extracted log data for 108 days from a single system where we had no prior knowledge of failures. The analysis of this data is presented in Section VI.

III. Analysis Goals

The initial focus of our study was to understand whether any useful information can be gleaned from the debug/trace logs of CM. We had three main goals in mind while doing the analysis.

Signature of a Log: The first question to ask was whether we could determine the signature of a log file under normal operating conditions. Systems vary widely in terms of size and functionality. In turn, the log files from different systems can differ greatly. We define the *signature* of a log file by (a) the type of messages that occur in the log, (b) the frequency of each message type, and (c) the distribution of the message types over time. As an example, a particular CM deployment may have periodic audit turned on and the signature of the resulting log should contain periodic occurrences of the audit messages. Note that identifying the signature of a log file under normal operation is also very important in identifying and predicting failures, as any deviation from the normal signature would be an indication of an anomaly or a failure.

Categorizing Failures: Often, the system logs contain descriptive information about a failure. This could be in the form of events leading up to the failure, and/or footprints from the recovery process. Each failure may have its own characteristic footprint. The ability to automatically derive this footprint from the logs, will help the system administrator in troubleshooting failure conditions.

Predicting Failures: The ultimate goal of the analysis is to be able to use these logs to predict and preempt failures. Previous studies have indicated that a system may go through a series of non-fatal errors before it really crashes [7]. There are many instances where a sequence of events directly leads to a failure in the system. By identifying the accompanying symptoms present in the log files, it is possible to predict upcoming failures reliably.

Analyzing the log data to achieve the above goals is a complex task. Some of the complexities of the analysis come from the following:

1) Log messages are often corrupted and/or lost, especially when the system is under failure conditions or under heavy load. Any log mining technique will suffer from the incorrectness that arises out of this.

2) The volume of data generated in the logs can be really large. Log messages are textual and any data mining tool will need to reduce the volume of data stored and processed, before the data can be handled in an efficient way.
3) The content of the log messages is very domain/application specific and detailed knowledge of the application and domain is needed to determine if a particular behavior is really anomalous. Any general purpose log analysis technique cannot deal with the application-specific nature of the logs.

Unfortunately, it is difficult to avoid the first problem of lost messages. Logging is usually the lowest priority task and an overloaded system may delay and/or lose log messages. However, our study indicates that there are enough indicators in the log file and losing a few has minimal impact on our ability to detect problems. In the next section, we discuss log preprocessing techniques that aim to purge the corrupted messages that arise out of the first problem and tackle the second problem through data reduction. Solution to the third problem is proposed in Section V-D.

IV. Log Preprocessing

In the CM application, several processes and threads write to the same log file. Even with locking, there are cases where the log messages are either partially written and/or corrupted. The first preprocessing step was to cleanup the log file by removing all corrupted messages and unprintable characters.

As shown earlier, the format of entries in the log file was Date:Time:Sequence:ProcessName(PID):Priority:[Payload]. By manually looking through the log file, we realized that most of the semantic information in the message was stored in the process name and the payload field. We define the message formed by combining these two fields *ProcessName:[Payload]* as the *Effective Message*. Henceforth, when we refer to *message*, we mean the effective message.

In order to determine the signature of a log file with respect to various message types, the first order of business was to identify all unique messages in the log files. We extracted out the effective message part from all entries of each log file and built a set of unique messages across all 714 logs. From the set of all **11.5 million** messages in the logs, there were about **2.5 million** such unique messages. In statistical terms, defining the signature of a log in terms of 2.5 million messages is like studying a 2.5 million variable space. This is not only very computationally expensive but may also lead to detection of no event patterns in the logs, since a large fraction ($\sim$ 22%) of the messages are unique. Therefore, before any analysis technique could be applied, we needed to reduce the unique message set through some form of clustering.

A. Message Clustering

Note that the payload part of the message and hence the effective message comes from debug/trace messages written by the system developers. Such messages are usually composed of a text string with some parameters. The parameters could be IP addresses, memory locations in the code, or any other generic alphanumeric strings.

Each invocation of a given message in the code may print out in the log file as a different message, depending on the values of the parameters in the message. One approach that could be taken to reduce the number of unique messages in the data set is to *de-parameterize* the messages and cluster similar messages together. The simplest and most coarse-grained de-parameterization technique that we used in the first round of analysis was to replace the following with generic tokens: (a) IP/Ethernet addresses, (b) memory locations, (c) all hexadecimal digits. Once this was done, we then clustered the messages based on Levenshtein distance [12]. This simple combination of de-parametrization/clustering reduced the number of unique messages to nearly **13,000** which was about 0.5% of the original message set. This is a more manageable set of information and the ratio of the unique messages to the total number of messages is small enough to detect possible patterns in the logs.

Note that there is a trade-off between the granularity of de-parameterization/clustering and the amount of information retained. A general purpose clustering technique like the one we used leads to a higher loss of information. However, the size of the unique message set is reduced significantly and becomes easier to analyze. After the first round of analysis, custom clustering/de-parameterization techniques (determined through domain-specific knowledge) can be applied to important subsets of messages to retain more information during the analysis.

B. Message Normalization

Storing textual messages in memory during analysis is very resource intensive and can slow down the analysis significantly. To avoid this problem, we normalized our message set by assigning unique numerical identifiers to each message in the set of all 13,000 unique messages. The techniques for assigning identifiers to messages has been discussed in our earlier work [13]. Henceforth, we will refer to these numerical identifiers as *message codes* or just *codes*. We pre-processed each of our 714 log files, such that each line of log was converted to a timestamp and a numeric code.

Next, we discuss the analysis of these pre-processed logs files.

V. Analysis for Logs with Known Failures

Our study was targeted towards two types of data sets. In the first data set of 714 logs, each log was retrieved from a system after a crash failure and had a distinct failure marker. These logs contained three hours of data before failure and one hour after failure. There were different types of crash failures in the logs and each was identified with the presence of its corresponding message code in the log. Our analysis for these logs was aimed towards finding a common signature across all failures of the same type and determining if the failure can be predicted through a build-up of messages. Note that even though we applied our techniques to all 714 log files, results

from a sample set of log files presented here are enough to showcase the validity of our techniques.

A. Frequent Itemset Mining

The first effort at analyzing the pre-processed logs was to detect common pattern across log files with similar types of failures. We wanted to find item sets that were commonly associated with each failure code. The hope was that the presence of these item sets would predict or at least characterize the failure. We grouped the logs based on their corresponding failure codes and mined for frequent item sets within the group. The high support frequent item sets turned out to be codes associated with the failure and recovery process. *Our results show that frequent itemset mining was a useful approach for categorizing failures as it indicated the set of common messages that occurred during the failure of a certain type.* However, it did not help in predicting failures.

As one may expect, the failure and recovery messages mostly occur just before and immediately after the failure. To eliminate them from our prediction analysis, we cut the logs just before failure, and perform frequent item set mining on the shortened log. We found that codes 4514 (related to process errors) and code 4597 (generated from the maintenance system) were the most common codes before failure.

4514 `hmm:CM_proc_err: pro=(NUM),err=(NUM),seq=(NUM),da=(NUM),...`

4597 `hmm:MTCEVT ERR type=(NUM) lname=(NUM) pn=...`

Unfortunately, their presence alone did not make good indicators of upcoming failure as they were fairly prevalent codes across all log files. However, the results here did prompt us to look deeper into these codes in subsequent analysis. This is discussed further in Section V-D.

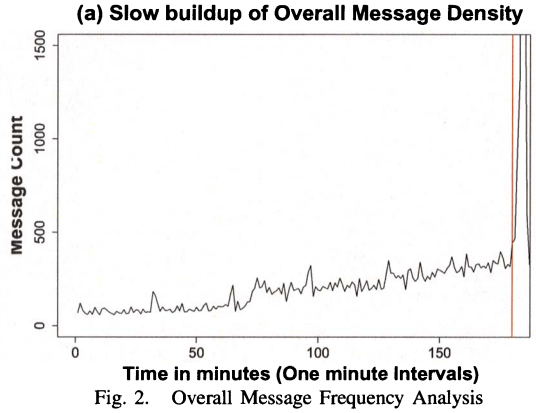

Fig. 2. Overall Message Frequency Analysis

B. Overall Message Frequency Analysis

In our earlier work [13], total message frequency (i.e. total number of message across all codes per unit time) was analyzed, showing some promise of information. This technique relied on the fact that processes become more chatty as the failure approaches, especially in the cases where there is a slow build up of failure in the system. As an example, Figure 2 shows the overall message frequency plotted against time. Message frequency was obtained for each 1 minute interval. The graph shows a steady increase in the message frequency before the failure at the 180 minute mark. In many cases simple visualization of the overall message frequency graphs helped in identifying common failures across log files.

C. Individual Message Frequency Analysis

Taking the next step, we looked at the message frequencies of individual codes in a log file. We wanted to find codes that showed peculiar trends before failure. As explained before, we focus our attention on the part of the log just before failure. We computed message frequencies for 1 minute intervals for each code and plotted these for visualization. A sample graph is shown in Figure 3(a). This is a graph of individual message frequencies for the same log file shown in Figure 2. There are 29 different message codes and except for a few dominant codes, it is difficult to visually determine any pattern in the other codes. We encountered similar difficulties across other log files where the number of codes was too large to explore. Therefore, we needed a mechanism to filter out only the "interesting" codes.

We implemented 3 types of automated filters to pick out the codes to plot: *slope*, *window-max*, and *window-rate* filters. These filters were targeted to bring out the codes that show some form of trend or anomaly before the failure.

Slope Filter: The slope-filter was the most natural. We are interested in identifying codes that steadily builds up towards the failure. We simply ran a linear regression on the individual message code frequencies and plotted only the interesting codes with slope higher than a specified threshold.

The slope filter captured codes 4514 and 4597. Figure 3(b) shows the slope-filtered graph for the same log shown in Figure 3(a). As can be seen from the graph, the number of "interesting" codes go from 29 down to 2. The graph shows that these two codes were largely responsible for the increased message frequencies detected in Figure 2. This behavior of codes 4514 and 4597 was observed in log files spanning across a number of systems. *Hence, for such systems, to do failure prediction based on a steady build-up of log messages, we only had to look at codes 4514 and 4597.*

Window-Max and Window-Rate Filters: The other two filters are based on the idea that there is a period of time just before failure during which the log behaves anomalously. Here we designate a specified window before failure as the window of interest and call it the *pre-failure window*. We would like to detect codes that had increased activity in this window. In our analysis, we used a pre-failure window of one hour. The window-max filter plots only the codes whose maximum frequency in the pre-failure window is a threshold-fold higher than the *maximum* frequency before the window. The rate filter plots only the codes whose average frequency is a threshold-fold higher than the average frequency before the pre-failure window.

The window-max and window-rate filters found more codes of interest than the slope based filters. It is worth noting that the two window based filters had similar results. This is due to the fact that a large number of codes only appear

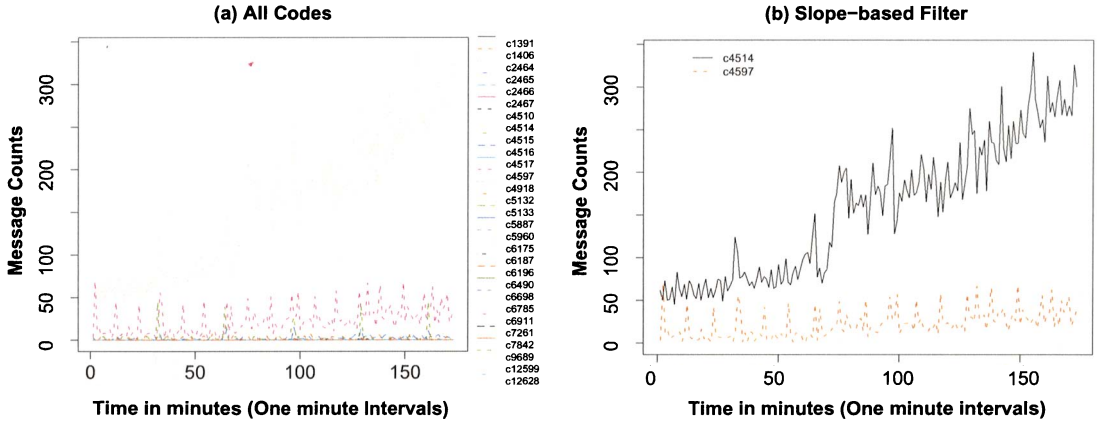

Fig. 3. Use of Slope-based Filters reduces codes

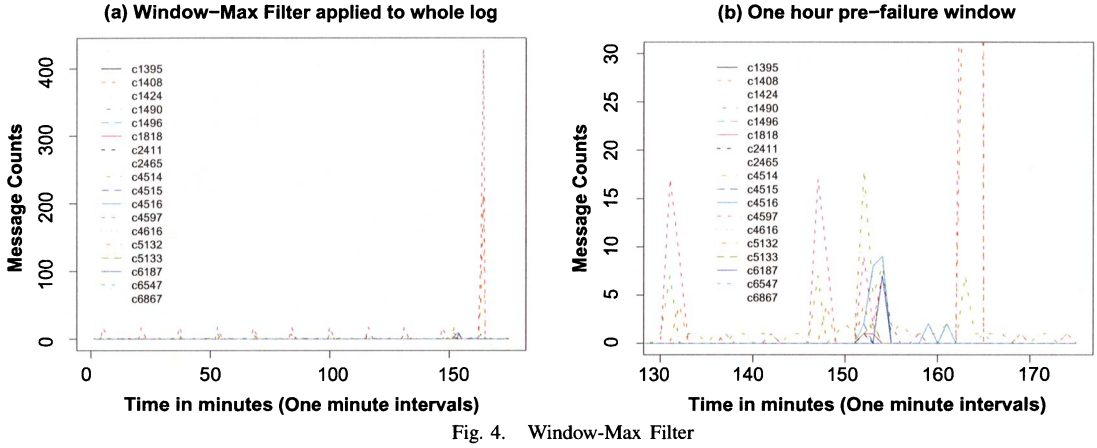

Fig. 4. Window-Max Filter

in the pre-failure window and not before. *This is evidence that there is information, other than the failure code itself, in the log files describing the failure.* By using the filters, we can identify these codes to help categorize the failure. For example, Figures 4(a),(b) show a set of codes obtained after applying the window-max filter on a log file from a system which had periodic failures over a course of time. Each failure of the system was found to have a signature similar to Figure 4(a). Figure 4(b) shows only the pre-failure window from Figure 4(a). Looking at the corresponding messages in the log file revealed that the set of messages were caused by a number of phones repeatedly trying to register themselves with the CM (an anomaly condition). In another case, there was a set of 48 logs that displayed highly similar codes related to scheduled maintenance just before failure. The failure turned out to be maintenance triggering a crash. The two examples given above show how filters can be used to extract failure signatures from the log files.

D. Application Specific Clustering

The results in Section V-C prompted us to look deeper into codes 4514 and 4597 that show an increasing trend towards failure. Recall that during the log preprocessing stage, we had de-parameterized and clustered the original log messages. We inevitably lost some information from the logs but this was a necessary step to reduce the total number of messages.

Now, we introduce the concept of *application-specific clustering*, where we allow the de-parameterization and clustering to be tuned to the specific application logs. As an example, for code 4514 below, we tune the clustering to not de-parameterize the first and second NUM fields, i.e. we reinstate some of the parameters in the 4514 messages to recover some of this information.

Original Clustered Message for code 4514:

```
hmm:CM_proc_err: pro=(NUM),err=(NUM),seq=(NUM),da=...
```

Application Specific Clustering:

```
hmm:CM_proc_err: pro=7171,err=200,seq=(NUM),da=...
hmm:CM_proc_err: pro=7172,err=603,seq=(NUM),da=...
```

This led to a breakdown of about 400 different sub-codes of the original code 4514. Extracting just these messages from the logs, we employed the techniques shown in Section V-C on this message set. The results showed that, while the codes may differ between releases, only a few codes in each release were responsible for the increasing trend found earlier. *This*

allowed us to isolate the exact error numbers and processes that were associated with the pre-failure increasing trend. Figure 5 shows the same log file as Figure 3 with slope-based filter applied on the sub-codes of code 4514. The graph shows only 2 of the 400 sub-codes were responsible for the increasing trend.

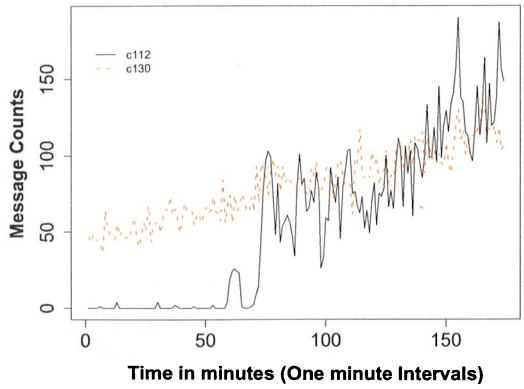

Fig. 5. Figure 3(a) with application-specific clustering of dominating code

VI. Analysis in the Absence of Failure Markers

In the above analysis, we have assumed that we know when the failures occur and are able to mark out the failure times. Note that without this information, none of the automated filters described in Section V-C make much sense. However, even in the absence of these failure markers, our techniques can still yield useful results. As an example, our second data set was a series of log files spanning a 108 days period, obtained from one CM deployment. Within this period, there were intervals when the network was very slow and impacted the quality of service provided by the CM. However, these times were not exactly marked in the logs through some known failure codes.

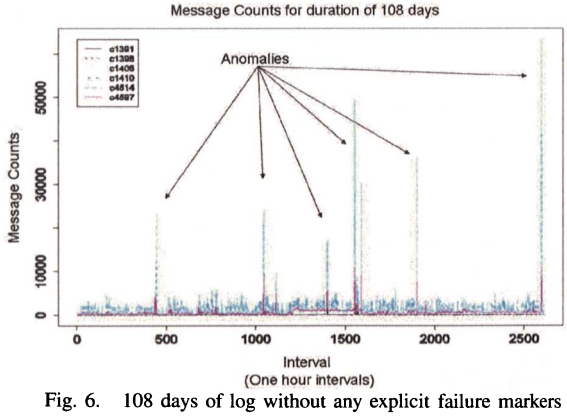

Fig. 6. 108 days of log without any explicit failure markers

In the absence of failure markers, we first plotted the overall message frequency and then used our knowledge gained from the previous analysis to track the usual suspect codes (e.g. 4514 and 4597). Both of these plots indicated anomalous regions. The overall message frequency plot showed intervals of time (about 1 hour) where the message frequency increased tenfold from normal, indicating possible problems. We zoomed into those areas and identifying them as "failure" areas, used techniques discussed earlier to get to the interesting codes. By plotting these interesting codes across the whole 108 days period, we see that they indeed corresponded to the dates when the network experienced difficulties. Figure 6 shows the plot of the message frequency per hour for the anomalous codes. Large sections of log file (e.g. first 450 hours $\sim$ 18 days) show normal behavior with sudden spikes in the codes where the network anomalies occur. Most of the anomalous codes were associated with large number of phones re-registering and slowing down the CM.

VII. Conclusions

Despite the many innate difficulties in dealing with the CM log files, we were able to extract meaningful results. The key to our approach is to transform the chaotic log files into a standard form for visualization. Together with user feedback and simple analysis tools, visualization allowed expert knowledge to be efficiently applied to anomaly prediction, detection and categorization. Our analysis techniques led to the detection and categorization of several failure types and in some cases predicted trends which lead towards failures.

The techniques and analysis presented in the paper need not be restricted to CM log files or even general log files. Any indexed (time or otherwise) set of text files can be visualized the same way.

References

[1] F. M. Facca and P. L. Lanzi, "Mining interesting knowledge from weblogs: a survey," *Data Knowl. Eng.*, vol. 53, no. 3, pp. 225–241, 2005.

[2] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam, "Critical event prediction for proactive management in large-scale computer clusters," in *Proc. ACM Int. Conf. on Knowledge Discovery and Data Mining*, pp. 426–435, 2003.

[3] I. Lee, R. Iyer, and D. Tang, "Error/failure analysis using event logs from fault tolerant systems," in *Fault-Tolerant Computing Symposium, FTCS-21*, pp. 10–17, June 1991.

[4] T.-T. Lin and D. Siewiorek, "Error log analysis: statistical modeling and heuristic trend analysis," *IEEE Trans. on Reliability*, vol. 39, pp. 419–432, October 1990.

[5] S. G. Eick, "Visualizing online activity," *Commun. ACM*, vol. 44, no. 8, pp. 45–50, 2001.

[6] D. Tang and R. Iyer, "Analysis and modeling of correlated failures in multicomputer systems," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 567–577, 1992.

[7] F. Nassar and D. Andrews, "A methodology for analysis of failure prediction data," in *Proc. Real Time Systems Symposium*, pp. 160–166, Dec. 1985.

[8] S. Ma and J. Hellerstein, "Mining partially periodic event patterns with unknown periods," in *Proc. of International Conference on Data Engineering*, pp. 409–416, 2001.

[9] S. Ma, J. L. Hellerstein, and C.-S. Perng, "Eventminer: An integrated mining tool for scalable analysis of event data," in *Knowledge and Data Discovery Workshop on Visual Data Mining*, 2001.

[10] R. Vaarandi, "A breadth-first algorithm for mining frequent patterns from event logs," pp. 293–308, 2004.

[11] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Proc. Conf. on Dependable Systems and Networks*, June 2007.

[12] Levenshtein Distance, "http://en.wikipedia.org/wiki/levenshtein_distance."

[13] R. Vasireddy, S. Garg, N. Singh, and S. Yajnik, "Log transformation technique for failure analysis of large communication systems," in *Fast abstract, Proc. Conf. on Dependable Systems and Networks*, June 2007.