# Mapping Moving Landscapes by Mining Mountains of Logs: Novel Techniques for Dependency Model Generation

Mirko Steinle[‡], Karl Aberer[‡], Sarunas Girdzijauskas[‡], Christian Lovis[†]

[‡]Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland

[†]Geneva University Hospitals (HUG), Geneva, Switzerland

## ABSTRACT

Problem diagnosis for distributed systems is usually difficult. Thus, an automated support is needed to identify root causes of encountered problems such as performance lags or inadequate functioning quickly. The many tools and techniques existing today that perform this task rely usually on some dependency model of the system. However, in complex and fast evolving environments it is practically unfeasible to keep such a model up-to-date manually and it has to be created in an automatic manner. For high level objects this is in itself a challenging and less studied task. In this paper, we propose three different approaches to discover dependencies by mining system logs. Our work is inspired by a recently developed data mining algorithm and techniques for collocation extraction from the natural language processing field. We evaluate the techniques in a case study for Geneva University Hospitals (HUG) and perform large-scale experiments on production data. Results show that all techniques are capable of finding useful dependency information with reasonable precision in a real-world environment.

## 1. INTRODUCTION

### 1.1 Background and Problem Statement: Mapping Moving Landscapes

By now, it is generally recognized that web services are a great means for system integration in heterogenous and dynamic environments. Furthermore, distributed architectures with loose coupling allow for simplified reuse of business logic and, by their modularity, promise to be less sensitive to local problems in components. On the downside, problems encountered, such as performance lags or inadequate functioning, are often difficult to detect and diagnose. Failures tend to be related to the component's interactions rather than to implementation errors in one specific component. Hence, there is need for an automated support to quickly track failures, functional errors and performance degradations. This problem, known as *root cause analysis* or *fault localization*, has been addressed by many research projects and plenty of industrial solutions have been proposed. Most of them rely on a dependency model providing representation of the dependencies between system components (see the review paper [31]). Beyond being a support for both manual and automated fault localization, a dependency model has various useful applications including *fault detection* [12], *impact prediction* and service *availability requirements determination* and constitutes in itself a valuable documentation and support for architectural decisions.

For complex and fast evolving environments, e.g. constantly moving landscapes, obtaining dependency information to create a dependency model in the absence of detailed management data is in itself a challenging task. Manual tracking of the dependency structure is practically unfeasible. While the problem of computation of dependencies is reasonably well mastered for the networking layer, where management protocols (like SNMP) are widely supported, the situation is very different in the case of dependencies between high level objects such as applications, web services and databases. Unfortunately, only relatively few research results exist that are applicable to a heterogenous and mission-critical system. Most do either fail to capture the dynamic nature of dependencies because they rely on compile-time data such as configuration files or software repositories [7], or they are *intrusive* and require application or middleware instrumentation [20, 24, 23, 27, 10], respectively perturbation of system operation [5, 11]. Only recently some commercial products have entered the market addressing the problem with more or less success [8].

In this paper we address the problem of deriving dependency models dynamically from data obtained in runtime. We are aiming at large-scale and mission-critical environments, such as hospitals or banks. Therefore the solutions have to be non-intrusive, scalable and easy to implement and maintain. The solutions we will propose have been developed in a real-world context, the information system infrastructure of a large university hospital, which we describe next.

### 1.2 The Geneva University Hospitals Environment

The Geneva University Hospitals (HUG) is a consortium of hospitals in four campuses and more than 30 ambulatory facilities in the state, comprising more than 2000 beds, 5000 care providers, over 45000 admissions and 750000 outpatients visits each year. It covers the whole range of in- and outpatient care, from primary to tertiary facilities. The

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
*VLDB '06,* September 12-15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

HUG is the major public healthcare facility in the Geneva region and the near France. The in-house developed Clinical Patient Record (CPR) is used in the complete HUG and runs on more than 4500 PCs. More than 20000 records are opened every day, 7 days a week, around the clock with never less than 200 records accessed each hour. The system is used by around 3000 care providers from all functions, including physicians, nurses, medical secretaries, social care providers, physiotherapists, nutritionists, music therapists, etc. In addition, the CPR is used for many other purposes than care, such as admission clerks, billing, resource management, epidemiology, and clinical research, amongst others. It is closely linked to the hospital information system and various third-party solutions that solve specific clinical tasks such as a radiology information system or an intensive care clinical system.

The Service for Medical Informatics of HUG has started to use HTTP/SGML services in the early nineties, before the establishment of standards like SOAP, WSDL or UDDI. Today, the multi-tiered clinical system is not only service- but also notification-oriented. The clinical system is critic for direct patient care and must be available around-the-clock, 7 days a week, throughout the year. As discussed in section 1.1 tremendous advantages can be drawn from a dependency model for problem diagnosis and architectural processes. An attempt to document the interdependencies between the many components manually has failed – an automated detection of dependencies is thus needed. The existence of a centralized logging system used by virtually all of the predominant in-house applications and most of the third-party solutions that have been integrated generated the idea of using log-based techniques as a particularly attractive alternative to static solutions. At the time of writing about 10 million of log messages are recorded per day, demanding more than 1 Terabyte of storage per year.

### 1.3 Contributions: Mining Mountains of Logs

In this paper we present three *non-intrusive* and scalable techniques to discover component interdependencies at *runtime* by mining system logs. In a first method, logs are regarded as simple *activity statements* of a given log-source at a given time, the sole information that has to be provided in a structured way. This is a very weak requirement and fulfilled by virtually all logging protocols. The technique is based on a recently published algorithm for mining temporal patterns from event logs [25] that we have adapted for our specific purpose. For the second technique, we view logs again as a simple activity statement but now within the *context* of a *user session*. Here, some structure or external information source is needed in order to identify the session logs are belonging to. We then mine these log sessions with a statistical procedure that is used for collocation extraction from document collections in the natural language processing field [17]. Finally, we take into account the free-text, e.g. unstructured part of the messages and exploit the existence of a service directory to identify logs related to invocations. The last technique needs as structured information only the log-source, but makes some assumptions about the content of messages. The assumptions should however hold for most environments where communication is based on http/xml services with a directory system.

We have implemented all techniques and studied their performance in the production environment of the Geneva University Hospitals. To objectively measure and quantify the performance of our techniques, we compare them in a first step against a reference model that has been created in a collective effort with the help of numerous system experts, and improved by detailed analysis of differences between the manually created model and the output of the algorithms. We show that all techniques are capable of finding valuable dependency information with reasonable precision in a, naturally noisy, real-world environment. The third technique in particular turns out to be very reliable. In a second step, to account for the dynamic nature of runtime dependencies, we validate the first two techniques against the third one and study the influence of system load on their performance.

To the best of our knowledge, this work is the first one to investigate the use of logs as information source for accomplishing dependency model creation in detail and, while experiments on relatively small benchmark applications ([1, 11]) or subjective evaluation on a production system ([16]) have been reported, it is also the first one to provide a performance study in a real-world environment of considerable size with objective criteria.

We have identified mainly two approaches to automated dependency model generation that meet the requirements of being dynamic and non-intrusive that hold for any critical production system. Ensel, in [15, 16], has proposed a very interesting neural network based technique applicable on a great variety of data. Unfortunately, the neural network has to be trained in a supervised manner, a laborious process, making the practical application of the technique expensive. Agrawal et al. in [1] have designed techniques based only on information about *activity periods* of the monitored objects. The techniques are non-intrusive in the sense that they rely only on data that is often, but not always available from existing monitoring infrastructure. As the authors show, it is nevertheless intrusive in the sense that enabling monitoring usually adds additional load and slows the system down. We provide in section 2 a more elaborate discussion of related work.

### 1.4 Paper Organization

The remainder of this paper is structured as follows. In the next section, we review related research and existing commercial solutions for the automated generation of dependency models. Then, in section 3 we present the approaches we have designed or adapted and evaluated in the production environment of Geneva University Hospitals, as reported in section 4. We discuss our results and future work in section 5 and conclude in section 6.

## 2. RELATED WORK

### 2.1 Automated Generation of Dependency Models

Approaches to automated generation of dependency models can be classified according to their capturing dependencies using *static* or *dynamic* (runtime) information [2].

*Static* approaches, such as [7], rely on analyzing system configuration, installation data, or application code to compute dependencies.

Approaches that operate at runtime can be classified according to their degree of *intrusiveness* into the managed

system. Many works use some kind of instrumentation of the managed objects themselves and are thus highly intrusive (f.ex. [20, 24, 23, 27]). Other, somewhat less intrusive approaches such as [10] use instrumentation of middleware only.

Both, static or intrusive dynamic approaches are difficult and sometimes impossible to apply in heterogeneous environments built with different commercial applications. Nevertheless, instrumentation is likely to become a very powerful and viable approach as soon as standards, such as JMX [22] or, language independent, ARM [13], are widely supported.

An interesting approach that does not require code modification is taken by [5] and [11]. Operation of components is actively disturbed, revealing dependencies in error conditions and allowing for characterization of their criticality. The authors report good empirical results, but it is obviously a very delicate, often impossible affair to deliberately introduce faults into a production system.

Conscious of the value of non-intrusive techniques, Ensel, in [15, 16], has attempted to decide on the existence of a dependency between objects based on time series of measurements of their activity only. Data, like for example CPU-Usage or TCP/IP communication activity, is collected by a system of monitoring agents and the decision on dependency is taken by an artificial neural network. Because the approach uses fairly general data, it can be applied in various settings making it very attractive. Ensel also mentions the possibility of using logs as an activity measure, but doesn't report any experience with doing so. Unfortunately, the neural network has to be trained in a supervised manner, a laborious and delicate process, making the practical application of the technique expensive. Furthermore, a study of the technique's performance, be it of theoretical or empirical nature, has to the best of our knowledge not been published.

Agrawal et al. in [1] have designed techniques for both synchronous and asynchronous systems based on information about *activity periods* of the monitored objects. In the asynchronous, more difficult case, the authors observe that for SQL queries executed during EJB transactions, the delay between the start of a transaction and an independent query appears to be completely random, while the delay for a dependent query shows some typical values. To exploit this feature, one builds histograms of delays and performs a $\chi^2$ test to measure the deviation from a uniformly random distribution. Experiments show that accuracy and precision of the technique are inversely proportional to the degree of parallelism (number of users) in the system and performs well under low load. The technique is non-intrusive in the sense that it relies only on information that is often, though not always and in particular not at HUG, available from existing monitoring infrastructure. As the authors show, it is nevertheless intrusive in the sense that enabling monitoring usually adds additional load and slows the system down. It depends on the particular environment, whether logs or monitoring data on activity periods are easier to obtain and consolidate.

## 2.2 Log Mining

The problem of mining logs to find dependencies between objects might appear to be closely related to *workflow mining*, also known under the name of *business process mining*, which aims at constructing models of a process given logs of its execution (see cite [29] for a recent example). However, it differs in two important aspects from our task. First of all, in workflow mining it is assumed that each log tells us the specific execution of a specific process it belongs to, while system logs are usually content to identify their source, such as an application module. Even if it is possible to identify the user session a log belongs to, there is still no reason to assume that each session is produced by the same process model. Hence, the initial situation is much more chaotic for our task. Second, there is a difference in the model to be discovered. Commonly made assumptions, though reasonable in the context of workflow mining, do clearly not hold for a dependency model of a distributed system, nor do they seem fitting for a single user session. On the other hand, we are a priori not interested in an entire flow of execution and such tricky issues as mutual exclusion or repetition.

Other works, from the domain of system management, address problems that are closer to ours. [28] develops Hidden Markov Model (HMM)-based monitoring of syslog streams that permits online detection of anomalous log sequences, identifying emerging or disappearing patterns of logging behavior and finding offline dynamic correlations between events in case of failure. [26] examines how unstructured messages can be categorized using text mining techniques associated with HMM to take temporal information into account. They also propose visualization techniques. [30] is also mainly concerned with creating classes of messages comparing an algorithm from bioinformatics (Teiresias) to SLCT, an algorithm developed by Vaarandi [32]. [25] proposes a window-free detection of dependent events. [19] describes a methodology to automatically validate, complete, and construct *Event Relationship Networks*, which are used in the action-oriented analysis (AOA) paradigm for event management developed by IBM and presented in [21].

## 2.3 Commercial Solutions

A recent study [8] compares five vendor solutions that provide some dependency mapping functionality. Three of them rely completely on scanning configuration data, that is, take a strictly static approach with the limitations discussed above. A fourth one operates at the network level, combining traffic analysis and agent-less scanning of TCP/IP ports. The last product, winning the comparative study, places agents on machines that hook into kernel I/O, gathering information on processes communicating in real-time. All examined products use a kind of template library to identify mainstream business applications and well-known services and protocols. Nevertheless, in an environment with many non-standard solutions, a very thorough configuration effort is still needed, for example, by manually identifying what physical process or service uniquely identifies a custom application.

# 3. APPROACH TO DEPENDENCY MODEL MINING

In this section, we present the techniques to discover dependencies between system components by mining logs, that we have designed and evaluated.

### 3.1 Approach $\mathcal{L}_1$: Viewing Logs as an Activity Measure

**Motivation and Key Ideas.** Being aware of the value of an approach relying on very generic data, as also advocated in Ensel's work ([15, 16]), we have designed a technique relying on statistical techniques. In contrast to techniques relying neural networks as used by Ensel, such an approach does not require supervised training. Our approach is motivated by the observation that the activity of interacting objects is often correlated. As activity measure we can use the logging events. Indeed, at the very least, a log entry provides information about its origin and the time of its creation. The minimal semantic content of such a log entry is hence *source S has been active at time t*. For illustration, consider figure 1 that shows the logging activity of two interacting applications. The first one, `DPIFormidoc`, calls the second one, `DPIPublication`, to publish medical documents. One can clearly see that periods of high and low activity are correlated.

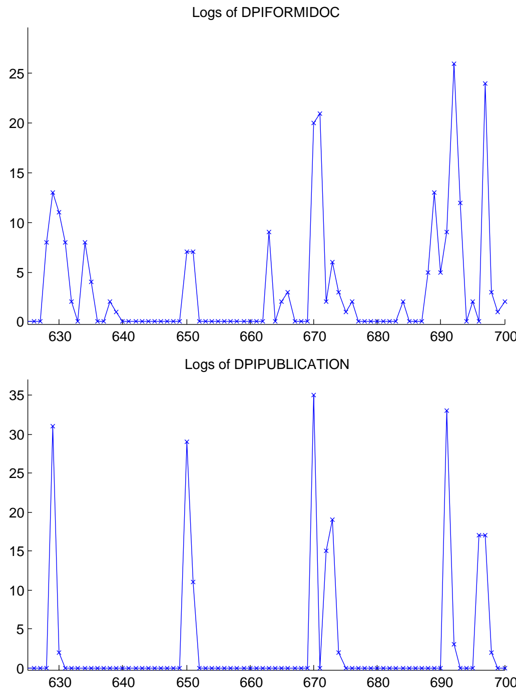


**Figure 1: Number of logs ($y$-axis) per second ($x$-axis) for two interacting applications**

Visual exploration of an important number of (dependent and independent) application pairs convinced us of the potential information content of logging activity data.

**Technique.** Let $A$ and $B$ be the sequences of timestamps of logs emitted by applications $App_A$ and $App_B$. Let the distance of a timestamp, or point, $t$ to a log sequence $A$ be the smallest absolute value of the differences between $t$ and any point in $A$, that is:

$$dist(t, A) = \min_{a \in A} |a - t| \tag{1}$$

We compare the *typical distance* of random points $r$ to $A$ to the typical distance of points $b \in B$ to $A$. More precisely, we create a sample $S_r$ of values $dist(r, A)$, where $r$ is random and a sample $S_b$ of values $dist(b, A)$, with $b \in B$ (subsampling of B). Then, we compute *confidence intervals* $CI_r$ and $CI_b$ for the *median* of respective samples using a robust order statistics method (explained among others in [9]) making as sole hypothesis independence. If the upper bound of $CI_b$ is below the lower bound for $CI_r$, we conclude that the logs of $App_B$ are closer to the logs of $App_A$ than random and we conclude that there is a dependence between the two applications.

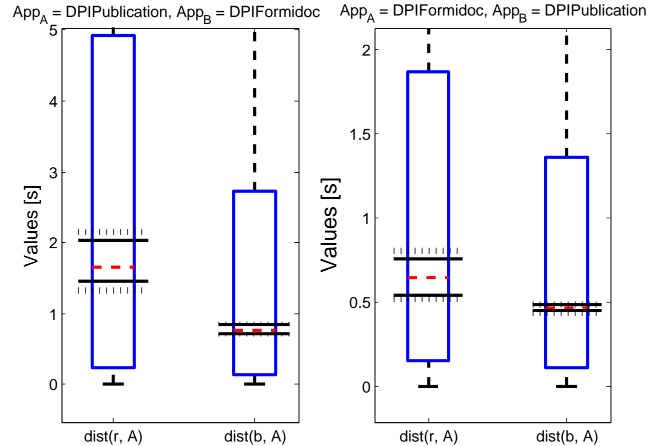


**Figure 2: Boxplots.**

For illustration, consider figure 2, which displays results obtained for the same data as in figure 1. The left graph shows two boxplots for the case where `DPIPublication` is playing the role of $App_A$ and `DPIFormidoc` the one of $App_B$. The first boxplot is for sample $S_r$ and the second for $S_b$. The dashed line is for the median, the solid lines for the upper and lower bound of the 95-level confidence interval, and the dotted lines for the bounds of the 99-level confidence interval. In the right figure the roles of the applications are inverted. In both cases, the upper bounds of both the 95 and 99 level confidence intervals for $S_b$ are below the lower bound of the confidence interval for $S_r$ and we would correctly conclude that the two applications are dependent.

The test is similar to the one developed by Li and Ma [25] as part of an algorithm for mining temporal dependencies between event types, which, in turn, has been inspired by results from geo-spatial statistics ([6]). Instead of a test for a difference of the mean, we use a robust test for the median. While the test is two-sided in [25] so as not to exclude temporal relationships of considerable length, our test is one-sided, e.g. we are testing if the median of $S_b$ is smaller than the one for $S_r$. Furthermore, while [25] considers the distance to the *next* arrival in $A$ from a given point $b$ in $B$, we consider the distance to the *nearest* arrival (equation 1).

**Underlying Variables and Scaling** When applying this test on longer intervals, one encounters an additional difficulty. Indeed, even applications that are not directly related expose some large-scale correlation due to their dependency on a common variable, which is the overall load of the system, itself dependent on *time*. Even though hospitals are working round the clock, there is still much more activity at usual office hours. A solution to this problem is to apply the test locally on periods that are sufficiently short to make the large scale dependency on time insignificant, and then

combine the local results into a global one. But then, it is important to have enough logs to make our test meaningful.

Based on this observation, we let $n$ be the total number of time slots, and the number $p$ of slots for which the test (with a 95 percent level confidence interval) is positive in both directions. We skip any slot where one of the two applications has fewer than *minlogs* logs, and call *support* $s$ the number of slots that have not been skipped. Let $pr$ be the percentage of positive tests among the possible time slots $pr = \frac{p}{s}$.

Then, to make the final decision, we define a threshold for the percentage of positive tests $th_{pr}$, as well as a support threshold $th_s$ and say a pair to be dependent if both $pr \geq th_{pr}$ and $s \geq th_s$.

### 3.2 Approach $\mathcal{L}_2$: Co-occurrence Statistics and User Sessions

**Motivation and Key Ideas.** The previous approach has used as little information from the logs as possible in order to preserve maximal generality. A main difficulty for the interpretation of logging data is the parallelism of the system, overshadowing the sequence of execution that could otherwise be learned from the logs by looking at their timestamp and source. Using structured information about the user and/or client machine at the origin of a transaction, we can identify logs that stem from the same *user session* and eliminate parallelism induced by the simultaneous activity of users. Parallelism due to asynchronous communication, however remains. The fact that both, a machine can be shared by different users, and a user might be active on different machines, makes session creation a challenging task.

Once sessions have been created, we still restrict ourselves to use only information on the log's source and time of creation. In other words, a session is treated as an ordered sequence of activity statements by different applications. A (very short) excerpt of a session is given in figure 3. There is one controlling application, $A_2$, corresponding to a lightweight client, that calls first $A_1$ and then twice $A_3$, which in turn calls $A_4$.

In many cases, it is quite easy to identify interactions from the log sequence, because, as in our example, logs of the caller usually immediately precede and/or follow logs of the callee. Intuitively, one expects that pairs of applications whose logs follow each other frequently are interacting. To give "frequently" a more precise meaning, we rely on co-occurrence statistics.

**Technique.** Because the problem of session creation is specific to our particular environment, we do not give the details on our algorithms here. The procedure for mining the sessions, once they have been created, includes two independent steps. First, one builds *contingency tables* for pair types. Then, a *hypothesis test* for association is performed on these tables. For the first step, we start by extracting all pairs of immediately succeeding logs from sessions and call a *bigram* the corresponding pair $(a, b)$ with $a$ and $b$ being the applications having authored the first, respectively the second log. We ignore bigrams where $a = b$. For our example session from figure 3, we would get, in that order: $(a_2, a_1)$, $(a_1, a_2)$, $(a_2, a_3)$, $(a_3, a_4)$, $(a_4, a_2)$, $(a_2, a_3)$, $(a_3, a_4)$, $(a_4, a_2)$. Then, we create a contingency table for each type $(A, B)$ of bigrams we have observed, which are in the running example $(A_1, A_2), (A_2, A_1), (A_2, A_3), (A_3, A_4)$ and $(A_4, A_2)$. For

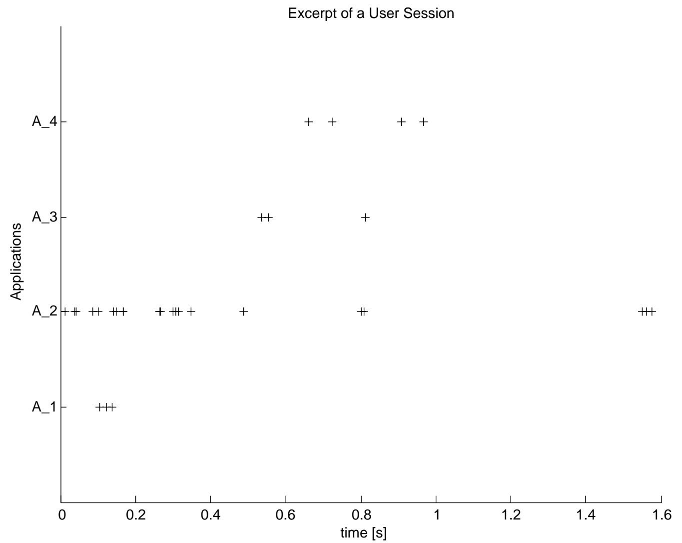


**Figure 3: An excerpt of a user session. $y$-axis: different applications, that is, log sources. $x$-axis: time in seconds since the timestamp of the first log.**

a given type, a contingency table provides a classification of all bigrams into four categories. Figure 4 illustrates this for the type $(A, B) = (A_2, A_3)$ in the running example.

| | $a = A_2$ | $a \neq A_2$ |
|---|---|---|
| $b = A_3$ | 2 | 0 |
| $b \neq A_3$ | 1 | 5 |

**Figure 4: Contingency table for bigram type $(A, B) = (A_2, A_3)$ in the running example (figure 3).**

In some cases, there is an important delay between two subsequent logs, for example, if they are triggered by different user actions. To deal with this, we introduce a *timeout*, that is, if the gap in time between some subsequent logs is higher than a given threshold, we do not use them to form a bigram. In our example, for instance, the last bigram $(A_4, A_2)$ would be ignored for any timeout value between 0 and 0.5 seconds.

In the second step, we apply a test for association on the contingency tables. Several well-known tests exist for that task. We opted for a test based on a log-likelihood statistics following asymptotically a $\chi^2$ distribution, which has been proposed by Dunning ([14]) and empirically shown to have a more desirable behavior for heavily skewed tables than the more common test by Pearson. The PERL implementation of the method uses UCS, a toolkit for the statistical analysis of cooccurrence data offered to the community by Stefan Evert [17, 18]. Terminology, too, has been taken from Evert's work.

### 3.3 Approach $\mathcal{L}_3$ : Analyzing Free Text

The third method is based on the following observation. Invocation of a remote service is an event an application's developer usually decides to log. Normally, the way of doing this is not standardized. To detect such logs one might try to search for keywords like *call*, *remote* or *invoke*, or to

apply more powerful algorithms such as those contributed by the authors' of [26]. But then, even if invocation logs are identified, we still need more processing of the contents to determine the specific service or method that has been invoked.

Alternatively, in the case of communication via web services based on a directory system, we can approach the problem the other way round. Indeed, although the detailed way a remote service invocation is logged is peculiar to each piece of code, respectively the code's author, it is extremely likely that some element provided by the directory system is mentioned in the log entry, as this kind of information is crucial for debugging and tracing purposes. Therefore, we can directly look for citations of directory entries in the free text part of a log and infer a dependency of the log's source on the directory entry it refers to.

At HUG, for mainly historical reasons, the directory in use is basically an XML file indicating the root URL of groups of functionally related services. All service groups have an identifier, as well as information related to replication issues.

Decision on a dependency is then straightforward: If, and only if, there are logs from application $A$ that are referring to service group $S$, $A$ is dependent on $S$.

For illustration, consider the invocation of a service `notify` belonging to the group `DPINOTIFICATION` and located on server `myserver` by some application. The free text field of a log of this interaction might look as follows:

```
Invoke externalService [fct [notify]
server [myserver.hcuge.ch:9999/myurl]]
```

Or, if the developer has decided to mention only the service group id:

```
(DPINOTIFICATION) notify( $myparams )
```

**Stop Patterns.** Often, a given call to server $S$ from a client application $C$ is not only logged by $C$, but also by $S$. To deal with the resulting problem of reversed dependency directions, we introduce *stop patterns*. A log that would otherwise be interpreted as a client's log is ignored if it matches the pattern.

# 4. CASE STUDY

We report in this section on the extensive empirical evaluations of the three different methods that were performed in the HUG environment.

## 4.1 Test Data

We have used data from 7 days, making a total of 56.8 mio logs. Given the constant evolution of the system, in order to facilitate the creation of a valid *reference model*, we have chosen days from a short a period as possible, i.e. a single week (see table 1).

| day [dec 05] | 06 | 07 | 08 | 09 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|
| #logs [mio] | 10.3 | 9.4 | 9.4 | 9.9 | 3.7 | 3.4 | 10.7 |

**Table 1: Days in test period with number of logs. $10^{th}$ and $11^{th}$ of December 2005 fall on a weekend.**

## 4.2 Clock Synchronization Issues

Both $\mathcal{L}_1$ and $\mathcal{L}_2$ rely on the temporal information provided by the logs. In HUG's logging system, log entries carry two timestamps that both have a resolution of 1 msec. The first one marks the creation of the message on client side, while the second one is defined by the log server on reception. Due to client-side buffering for performance reasons, we can not use the latter timestamp. Hence, we have to pose the question how clock synchronization is handled. Unix server clocks are synchronized via the standard Network Time Protocol (NTP) against external reference servers and can be expected to deviate less than 1 msec. Alternatively, the clocks of Windows NT servers, as well as those of client machines are only synchronized within their own NT domain and not against a reference clock. As for the NT servers, we have verified that deviation with respect to the Unix servers is less than 1 sec. In our experiments we use the timestamp values as they are, and do not undertake any correction or rounding and so the algorithms had to operate and work with slightly imprecise values.

## 4.3 Reference Model

To obtain a reference model we have meticulously consulted all available system experts and developers of the respective applications or services. For method $\mathcal{L}_1$ and $\mathcal{L}_2$ the model consists of pairs of *log sources* (applications in our case), which are said to be dependent if they are directly interacting. We do not consider the *direction* of the interaction here. For method $\mathcal{L}_3$ , the model is a set of pairs composed by an application (or log source) and a service directory entry this application is using. In the first model there are 54 applications, resulting in 1431 $((54^2 - 54)/2)$ different pairs, among which 178 are known to be dependent. In the latter one, we consider 52 applications and 47 service directory entries. 177 dependencies are known. The slight difference in the number of applications stems from the fact that we were forced to ignore a few cases where it has been difficult to obtain sufficiently precise information on the type of dependency considered. The closeness of the number of dependencies is a consequence of the fact that the mapping between applications and service directory entries is often one-to-one.

## 4.4 Strategy

In a first step, we perform experiments for methods $\mathcal{L}_1$ , $\mathcal{L}_2$ and $\mathcal{L}_3$ individually. We apply the techniques for each day independently, which allows us to quantify the accuracy of our observations by computing confidence intervals using the robust order statistics method from [9]. To validate the results of our algorithms, we use the reference model elaborated with the experts. Note, however, that the reference model is *static*, that is, they tell us about the potential existence of an interaction and not if it has really taken place, while the techniques are *dynamic* by nature. To solve this issue, in a second step, we use method $\mathcal{L}_3$ , which turns out to be very reliable in general, to evaluate the other two techniques on much shorter periods of one hour each and study the influence of the overall load the system is experiencing.

## 4.5 Results for Approach $\mathcal{L}_1$

We divided each day into 24 periods of one hour ($n = 24$) and skipped applications that have fewer than 100 logs in a given period ($minlogs = 100$). Results are given for threshold values $th_{pr} = 0.6$ and $th_s = 0.3$. Parameter values have been defined after preliminary experience with data lying outside the test period.

The algorithm detects between 30 and 46 true dependencies at the expense of between 11 and 22 false positive decisions (figure 5). A confidence interval for the median of the percentage of true dependencies among the positive decisions with level 0.984 is [0.63, 0.73]. While the number of false positives might seem high at first sight, the classification error in the case of unrelated pairs is actually low. Given that the test has been applied on 1253 unrelated pairs, a number of 25 false positives would result in an error rate of only 2%. On the other hand, it is clear that many dependent pairs have not been detected.

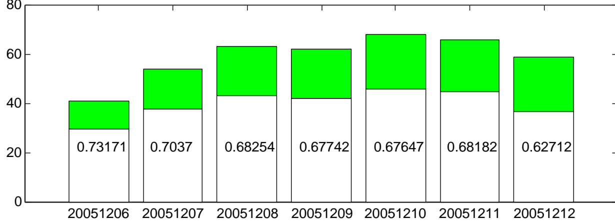


**Figure 5: Positive decisions for method $\mathcal{L}_1$ with $th_{pr} = 0.6$ and $th_s = 0.3$ for all days in observation period. Lower area: *true positives*. Upper area: *false positives*. Numbers: ratio of true positives.**

A detailed analysis of false positive decisions shows that virtually all of them are due to the difference between the semantics of correlation and interaction dependency, rather than to an erroneous detection of association between the logging activity. Indeed, in case of a transitive interaction dependency, or a frequent concurrent use of applications, it is obvious that the activity of the applications is correlated. This happens for example if the creation of a view in a GUI application requires to combine information provided by different components, such as laboratory results and administrative patient history.

## 4.6 Results for Approach $\mathcal{L}_2$

The session creation algorithm produced about 4000 sessions for week days and about 1000 on Saturday or Sunday. The percentage of logs that can be assigned to a session varied between 7.5 and 11% on the different days. With a timeout value of 1 second, co-occurrence analysis identified between 62 and 74 correct dependencies on week days, 51 on Saturday and 52 on Sunday, at the expense of between 21 and 25 false positives on week days and 19, respectively 21 for the week-end. The borders of the 98.4% level confidence interval for the median true positive ratio are 0.71 and 0.78.

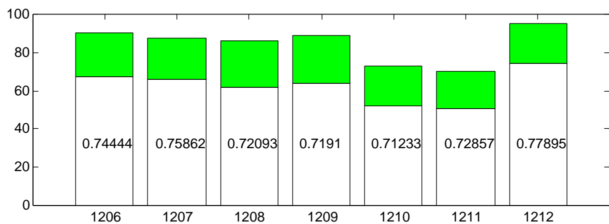


**Figure 6: Positive decisions for method $\mathcal{L}_2$ with $timeout = 1$ for all days in observation period. Lower area: *true positives*. Upper area: *false positives*. Numbers: ratio of true positives.**

While the creation of user sessions eliminates parallelism due to multiple users, it does not solve the issue of concurrency introduced by asynchronous communication. The analysis of false positives by manually decoding numerous sessions, shows that this kind of concurrency is clearly the major cause for that type of error. Indeed, all of the about 25 pairs we have analyzed turned out to be either transitively interacting or, more often, to be used concurrently in some frequently occurring situations where communication is asynchronous.

## 4.7 Influence of the Timeout

As explained in §3.2 we have introduced a *timeout* with the goal of improving the accuracy of the technique. Figure 7 shows the positive results for one day for different timeout values. We observe that the introduction of a timeout value that is neither too small nor too big increases the percentage of correct decisions among the positive ones. However, it also seems that the absolute number of correctly detected dependencies is slightly reduced as compared to not using timeout values.

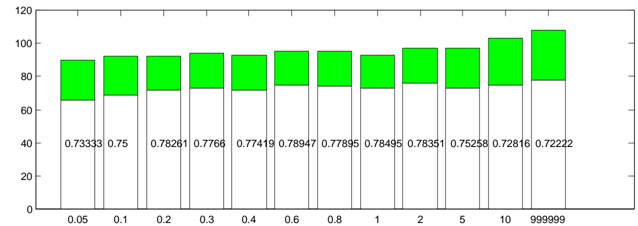


**Figure 7: Positive decisions for $\mathcal{L}_2$ on 12.12.2005 for different timeout values ($x$-axis, in seconds).**

To confirm this observation, we perform the following test: For all days in the observation period, we computed the true positive ratio $tpr_{to}$ and the absolute number $tp_{to}$ of true positives for timeout values $to = 0.3, 0.6, 0.8$ and 1.0 as well as $infinity$. Then, for each combination between a finite and the infinite timeout value, we do a median test for the difference of the respective values. That is, we compute a 0.98 level confidence interval and, if it is strictly positive, respectively negative, we reject the hypothesis of a zero or negative, respectively zero or positive median. In addition, we perform a signed wilcoxon rank sum test for all samples for a null hypothesis of zero median for the difference.

| | $tpr_{to} - tpr_{\inf}$ | $tp_{to} - tp_{\inf}$ |
|---|---|---|
| $to = 0.3$ | 5.4 (1.9, 9.3) | -7 (-13, -4) |
| $to = 0.6$ | 4.5 (2.0, 6.8) | -5 (-9, -3) |
| $to = 0.8$ | 4.5 (2.3, 5.7) | -4 (-8, -3) |
| $to = 1.0$ | 5.1 (1.7, 6.3) | -5 (-7, -3) |

**Table 2: Median values with confidence interval bounds (between parentheses) for timeout influences in $\mathcal{L}_2$ .**

The p-value of the signed wilcoxon rank sum test is 0.0156 for any two samples of size 7, such that the values of the one are always below the corresponding value of the other, and thus the same in all cases. Combined with the results depicted in table 2, for each timeout $to = 0.3, 0.6, 0.8, 1.0$, we have to reject both the hypothesis that its introduction does not increase the true positive ratio and the hypothesis that is does not decrease the absolute number of true positives.

In other words, while the introduction of a timeout reduces the total amount of positive decisions, it eliminates proportionally more false positives than true positives and can thus contribute to improve the quality of the algorithm's output.

### 4.8 Results for Approach $\mathcal{L}_3$

Applying the algorithm separately on all days in the observation period, we observe between 141 and 152 true positive decisions on week days and 116, respectively 117 on the weekend. With 10 stop patterns, the number of false positives is between 7 and 11 on week days and 5 on the weekend. A 0.984 level confidence interval for the median percentage of true positives among all positives is given by 0.93 and 0.96.

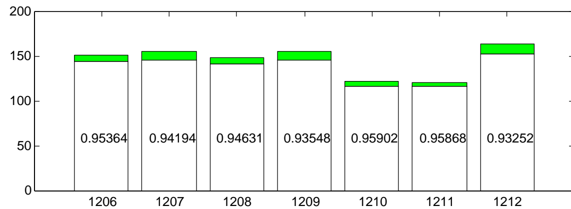


**Figure 8: Positive decisions for method $\mathcal{L}_3$ with stop patterns for all days in observation period. Lower area: *true positives*. Upper area: *false positives*. Numbers: ratio of true positives.**

On week days, between 28 and 39 known dependencies have not been discovered. To know which of them are really false negatives in the sense that the interaction did take place but has not been discovered, and which simply did not happen, we combine the results from all days and analyze false negatives in detail.

There are 16 false negatives in total (e.g. 161 dependencies have been detected). 6 of them are used extremely seldom according to the developers' knowledge, but are logged in a way that would allow their discovery. Therefore, we can conclude that they did not take place and hence are in fact *true* negatives. 7 interactions are not logged by the applications and 3 are logged but under a wrong name (for example, the service directory id UPSRV is used instead of the newer version of the same service UPSRV2).

False positives for all seven days, being the union of individual results, come in a number of 19. 2 of them are inverted dependencies caused by server side logs, 5 are transitive dependencies due to the log of an exception stack trace returned by the intermediary, 7 are due to coincidence (a patient having the same name as a given service id, for instance), and 5 are a consequence of the usage of a similar, but erroneous, service group id by the application. Without stop patterns, the number of inverted dependencies due to server side logs increases to 24.

### 4.9 Influence of the System's Load

In the preceding experiments, we observe that both, $\mathcal{L}_2$and $\mathcal{L}_3$ detect less dependencies on the weekend than on workdays, which reflects the real situation. Alternatively, method $\mathcal{L}_1$seems to detect more dependencies on the weekend. Intuitively, this can be explained by the fact that the higher the system's load, the more parallelism there is, which disturbs the analysis of association between logging activity. Method $\mathcal{L}_2$ should be sensitive only to the noise induced by concurrency in a given user session and not in the overall system. To verify the hypothesis that technique $\mathcal{L}_1$ performs better in periods of low load than in periods of high load, while $\mathcal{L}_2$is not affected by the system's load, we conduct the following experiment:

For each hour of all seven days, we use technique $\mathcal{L}_3$ to identify realizations of dependency relationships. We eliminate 4 applications which do not log all of their invocations to increase reliability of the output of $\mathcal{L}_3$ . Then, we compute the percentages $p_1$ and $p_2$ of these dependencies that can be found by methods $\mathcal{L}_1$and $\mathcal{L}_2$respectively. We use the number of logs as a measure of the system load and show the percentage of correctly identified dependencies as a function of the number of logs. We perform a linear regression and check if the confidence interval for the linear factor is strictly negative in case of $\mathcal{L}_1$ , respectively includes zero in the case of $\mathcal{L}_2$ . Furthermore, we plot the values of the variables as a function of time and visually inspect it.

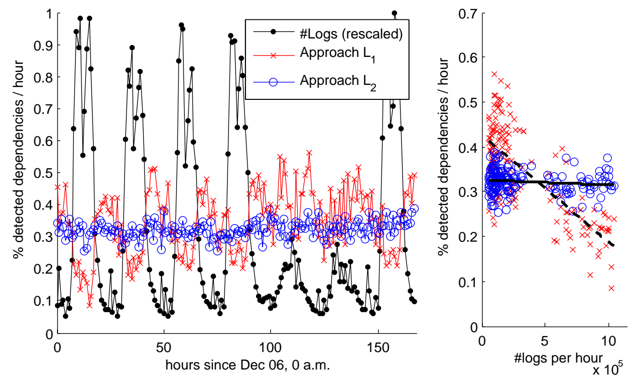


**Figure 9: Left graph: Number of logs (rescaled to fit between 0 and 1), $p_1$ and $p_2$ as a function of time. Right graph: $p_1$ (crosses) and $p_2$ (circles) as a function of the number of logs with regression lines (dashed for $\mathcal{L}_1$ , solid for $\mathcal{L}_2$ ).**

On the left graph in figure 9, we can clearly observe that the curve for $p_1$ behaves inversely to the curve for the number of logs, while such behavior is not evident in the curve for $p_2$. The confidence interval for the linear factor in the case of $p_1$ is strictly negative (delimited by -0.284 and -0.215), while it includes 0 in the other case (-0.025, 0.002). The validity of the regression model is verified by the means of normal qqplots for the residuals.

The percentage of false positives among all positives does not seem to be influenced by the system's load for both methods, as the confidence interval for the linear factor in the regression model includes zero in both cases.

### 4.10 Solution for HUG

As a result of our experiments we could show that in particular $\mathcal{L}_3$is a very effective solution for the automated generation of a dependency model of HUG's clinical system which satisfies the requirements of being completely non-intrusive and low-cost.

## 5. DISCUSSION AND FUTURE WORK

We will discuss the techniques according to four criteria: the quality of their *output*, the scope of their *applicability*,

their *impact* on performance and security of the managed system, and the amount of *effort* needed for implementation and maintainance.

Unsurprisingly, we can observe a negative correlation between scope of applicability and accuracy and precision of the output: based on very general information, $\mathcal{L}_1$ is applicable on virtually any type of logs, but its findings have been less precise than both those of $\mathcal{L}_2$ and $\mathcal{L}_3$ in the conducted experiments. $\mathcal{L}_2$ exploits and requires additional contextual information, and thus narrows the scope down. There are however plenty of settings imaginable where session information needs to be logged in order to have a complete trace of user activity, an online banking application for example. The best results have been obtained with $\mathcal{L}_3$ by taking into account the message of the log and using as additional information source the service directory. With the spread of SOA, more and more systems will fulfill the requirement of the existence of a service directory.

Naturally, any technique for automated dependency model generation should cause less damage than doing good, e.g. should have only minimal impact on the system's performance. In the same spirit, it should also save more time and management effort than needed to set it up and to maintain it. For log-based techniques, we need to distinguish between the collection and consolidation phase and the actual application of the algorithm. It is clear that the second phase should neither impact the system in a negative way, nor should it be difficult to implement and maintain. Note in particular that all algorithms scale linearly with respect to the number of logs. Collection and consolidation is more critical. Nevertheless, by making only little assumptions on the logs' structure, consolidation effort is kept low. As for collection of logging data from decentralized storage locations, it does not impact applications directly, but consumes some server resources nonetheless. Here, an important advantage is the possibility to interrupt collection in periods of high load without information loss.

In their current state, neither $\mathcal{L}_1$ nor $\mathcal{L}_2$ are able to discover the direction of an invocation dependency. This is a drawback, but we should not forget that the direction of an invocation not necessarily corresponds to the direction of the functional dependency. In the case of the frequently encountered *push* update scheme for instance, it is the callee that relies on the information provided by the caller, while the latter has a priori no need for the former in order to operate fine.

Nevertheless, it would be useful to improve our techniques to detect directionality. For $\mathcal{L}_1$ , we do have little hope, as neither visual exploration of the activity graphs did allow our human brain to draw any conclusion, nor did Ensel succeed with the artificial neural network approach. For $\mathcal{L}_2$, the problem stems from asynchronous communication semantics, as well as from the fact that callers usually log both, before and after an invocation. Given a dependent pair type $(A, B)$, one could try counting the number of times the first element of the *first* pair of the given type is an instance of $A$, respectively $B$, in a sequence of logs that is not interrupted by a pause of at least the length of the *timeout* parameter.

Another direction for improvement is to apply algorithms like the ones presented in [1, 3, 25] to analyze *typical delays* between logs. In case of $\mathcal{L}_2$ , this might help to distinguish frequent co-occurrences due to concurrency from those that are causally related.

There are at least two approaches to improve the handling of the stationarity issue in $\mathcal{L}_1$ . First, one could create time slots adaptively by measuring the degree of stationarity with existing statistical tests. Second, instead of comparing the distance to $B$ of logs in $A$ with a homogenous process, we could use a non-homogenous process whose intensity is proportional to the total number of logs. In addition, works like [4] propose more rigorous, but computationally expensive, ways of testing locally for global association of point processes. One could also study the benefit of classifying log messages of a given application in a preprocessing step, using algorithms mentioned in §2.2. Last but not least, we are currently evaluating techniques based on packet capture and protocol analysis.

# 6. CONCLUSIONS

We have developed non-intrusive and scalable techniques to discover dependencies between components of a distributed system by mining logs. An evaluation in a complex real-world production environment has shown that all of them provide useful results, with a performance that is proportional to the amount of semantic content of log messages considered.

**Acknowledgements** First of all, we would like to thank Alexander J. Lamb, head of the group developing HUG's logging system, for skillful management of the organizational aspects of the project, and, together with Florian Fischer, for interesting lunchtime-talks on related and unrelated matters. Sincere thanks go to Gilles Cohen, Jeremy Llewellyn and Michael I. Schumacher for reviewing and inspiring discussions. We gratefully acknowledge the diverse contributions of numerous members of Geneva University Hospital's IT staff, in particular to the creation of the reference model, including, but not limited to, Pierre-Antoine Arnet, David Bandon, Jérôme Billet, Eric Burgel, Nicolas Cassoni, Emmanuel Durand, Florian Fischer, Arnaud Garcia, Christian Girard, Dominique Guérin, Damien Grauser, Monique Long, Laura Remondino, Georges Robin, Frédéric Rybkowski, Paul Seed, Stéphane Spahni, Jean-Christophe Staub, Sébastien Tuet, Julien Vignali, and Anne-Marie Zogg.

# 7. REFERENCES

[1] Manoj K. Agarwal, Manish Gupta, Gautam Kar, Anindya Neogi, and Anca Sailer. Mining activity data for dynamic dependency discovery in e-business systems. Technical report, IBM Research Division, 2004.

[2] Manoj K. Agarwal, Manish Gupta, Anindya Neogi, and Gautam Kar. Discovering dynamic dependencies in enterprise environments for problem determination. In Marcus Brunner and Alexander Keller, editors, *DSOM*, volume 2867 of *Lecture Notes in Computer Science*, pages 221–233. Springer, 2003.

[3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP'03*, October 2003.

[4] Denis Allard, Anders Brix, and Joel Chadoeuf. Testing local independence between two point processes. *Biometrics*, 57:508–517, June 2001.

[5] S. Bagchi, J.L. Hellerstein, and G. Kar. Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment. In *12th Intl. Workshop on Distributed Systems: Operations & Management*, 2001.

[6] Mark Berman. Testing for spatial association between a point process and another stochastic process. *Appplied Statistics*, 35(1):54–62, 1986.

[7] U. Blumenthal, G. Kar, and A. Keller. Classification and computation of dependencies for distributed management. In *Proceedings of the Fifth IEEE Symposium on Computers and Communications (ISCC)*, pages 78 – 83, July 2000.

[8] Bruce Boardman. Map quest. `http://www.networkcomputing.com/showitem.jhtml?articleID=169600209`, September 2005.

[9] Jean-Yves Le Boudec. Performance evaluation of computer and communication systems. Swiss Federal Institute of Technology Lausanne, March 2005. Version 2.0.Beta.1. Available at `http://icalwww.epfl.ch/perfeval`.

[10] E. Brewer, M. Chen, A. Fox, E. Fratkin, and E. Kiciman. Pinpoint: Problem determination in large, dynamic, internet services. In *International Conference on Dependable Systems and Networks (IPDS Track), Washington D.C.*, 2002.

[11] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Seventh IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[12] Haifeng Chen, Guofei Jiang, Cristian Ungureanu, and Kenji Yoshihira. Failure detection and localization in component based systems by online tracking. In *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 750–755, New York, NY, USA, 2005. ACM Press.

[13] OpenGroup consortium. Systems management: Application response measurement (arm). OpenGroup Technical Standard C807, UK ISBN 1-85912-211-6, 1998.

[14] Ted Dunning. Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics*, 19(1):61–74, 1993.

[15] C. Ensel. A scalable approach to automated service dependency modeling in heterogeneous environments. In *Proceedings of the Fifth IEEE International Enterprise Distributed Object Computing Conference*, pages 128–139, Seattle, WA, US, 2001.

[16] C. Ensel. *Abhaengigkeitsmodellierung im IT-Management: Erstellung eines neuen, auf Neuronalen Netzen basierenden Ansatzes*. PhD thesis, Ludwig-Maximilians-Universitaet Muenchen, 2002.

[17] Stefan Evert. *The Statistics of Word Cooccurrences: Word Pairs and Collocations*. PhD thesis, University of Stuttgart, 2004.

[18] Stefan Evert. www.collocations.de. Web resources for collocation mining and coocurrence statistics, 2005.

[19] G. Grabarnik, J. Hellerstein, S. Ma, C.-S. Perng, and D. Thoenen. Data-driven validation, completion and construction of event relationship networks. In *Proceedings of SIGKDD'03*, 2003.

[20] Peer Hasselmeyer. Managing dynamic service dependencies. In *12th International Workshop on Distributed Systems: Operations & Management (DSOM 2001), Nancy, France*, pages 141–150, 2001.

[21] J.L. Hellerstein, J. Riosa, and D. Thoenen. Event relationship networks: A framework for action oriented analysis for event management. In *International Symposium on Integrated Network Management*, 2001.

[22] Java management extensions (jmx). `http://java.sun.com/products/JavaManagement/`.

[23] M.J. Katchabow. Making distributed applications manageable through instrumentation. *Journal of Systems and Software*, 45, 1999.

[24] F. Kon and R.H. Campbell. Dependence management in component-based distributed systems. *IEEE Concurrency*, 8(1):26–36, 2000.

[25] Tao Li and Sheng Ma. Mining temporal patterns without predefined time windows. In *ICDM*, pages 451–454, 2004.

[26] Tao Li, Sheng Ma, and Wei Peng. Mining logs files for computing system management. *SIGKDD Explorations*, 2005.

[27] V. Machiraju, J. Ouyang, A. Sahai, and K. Wurster. Message tracking in soap-based web services. In *Proceedings of the Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pages 33– 47, 2002.

[28] Yuko Maruyama and Kenji Yamanishi. Dynamic syslog mining for network failure monitoring. In *Proceedings of SIGKDD '05*, 2005.

[29] Ricardo Silva, James G. Shanahanand, and Jiji Zhang. Probabilistic workflow mining. In *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 275–284, New York, NY, USA, 2005. ACM Press.

[30] Jon Stearley. Towards informatic analysis of syslogs. In *Proceedings of IEEE International Conference on Cluster Computing*, September 2004.

[31] M. Steinder and A. Sethi. The present and future of event correlation: A need for end-to-end service fault localization. In *Proc. IIIS SCI: World Multi-Conf. Systemics Cybernetics Informatics, Orlando, FL*, 2001.

[32] Risto Vaarandi. A clustering algorithm for mining patterns from event logs. In *Proc. of 2003 IEEE Workshop on IP Operations and Management (IPOM2003)*, 2003.