A Logging Approach for Effective Dependability Evaluation of Complex Systems

M. Cinque*, D. Cotroneo*, A. Pecchia*†
*Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II,
Via Claudio 21, 80125, Naples, Italy
†Laboratorio CINI-ITEM "Carlo Savy", Complesso Universitario Monte Sant'Angelo, Ed. 1,
Via Cinthia, 80126, Naples, Italy
{macinque, cotroneo}@unina.it, antonio.pecchia@consorzio-cini.it

***Abstract*—The dependability evaluation and management of complex systems is often based on the collection of field data from event logs. Nevertheless, key decisions about log production and management are usually left to the late stages of development, leading to heterogeneous, inaccurate, and redundant logs. This in turn decreases the level of trust on logs. This paper proposes to enrich traditional logging by defining a set of rules, to be followed at design time, specifically conceived to improve the quality of logged failure data and to ease the coalescence of redundant or equivalent data. A tool for processing our rule-based logs has been developed to show the feasibility of the approach. The tool is applied on a real-world case study in order to evaluate the effectiveness of the approach when compared with traditional logging.**

***Index Terms*—Dependability Evaluation, Field Failure Data Analysis, Logging Rules, Automated Log Analysis.**

I. Introduction

Field Failure Data Analysis (FFDA) is a widely adopted methodology for the dependability evaluation of computer systems. It consists in observing spontaneous occurrences of failures in a system during its operational phase, without forcing or inducing artificial failures. One of the most adopted sources of failure data is represented by event logs. Logs are a collection of files where applications and system modules register their normal and anomalous activity, in the form of events. For this reason, "logs are the first place where system administrators go when alerted to a problem, since they are one of the few mechanisms for gaining visibility of the behavior of the system" [1].

However, in most of existing complex systems, logs are a quite under-utilized resource, because of their *unstructured* nature [2] and subjectivity [3]. The lack of a widely accepted strategy to log events compromises the potential of the collected field data. As a matter of fact, logs production and management is left to designers and programmers, each of them with his own experience and attitude. In other terms, crucial decisions about logging are left to the last steps of the development cycle. As a result, logs may be heterogeneous and inaccurate [4], [5]. Heterogeneity, which increases as the system complexity increases, may affect both format and content. While format heterogeneity is addressed by many current FFDA tools, content heterogeneity is more challenging since the meaning of a logged event depends on what the developer intended to log. On the other hand, inaccuracy is related to the presence of duplicate or misleading entries or to the absence of important failure data. This in turn compromises the ability of discriminating events related to actual failures from presumed ones.

Another threat comes from the presence of failure propagation phenomena, resulting in multiple and apparently uncorrelated events in the logs. A widely used strategy to remove this phenomena is to adopt *one-fits-all* temporal window to coalesce events. However, this is usually done without any awareness of the actual correlation among messages [1]. The risk is to classify correlated failures as uncorrelated, and viceversa, leading to results which do not necessarily reflect the reality.

We claim that a promising solution to overcome these limitations is to re-think the way in which logs are produced. In other words, we aim at improving how failure data is generated in order to significantly reduce analysis efforts. The ultimate objective is to build logs which allow to (i) unambiguously detect the occurrence and the location of failures (in particular, in this paper we focus on timing failures [6], e.g., component crash or hang) , and (ii) trace failure propagation phenomena induced by interactions within the system.

To this aim, we preliminary define a minimal set of logging-rules to be followed at design and development time. These rules are inspired to a high-level model of the entities composing a system and of the interactions among them. We show how to process rule-generated events to effectively pinpoint failure locations, and to discriminate local failures from propagated ones. A support tool has been also developed to automatize the processing phase. An experimental evaluation is finally provided on a real-world case study. A comparison with the results ideally achievable with traditional logging techniques shows how the proposed approach leads to more effective FFDA results.

II. Related Work

Log files are usually conceived as human-readable text files for developers and system administrators to gain visibility in the system behavior, and to take actions in the face of failures. Through a simple programming interface, applications can write events (i.e., lines of text in the log) according to developer's needs. Well known examples of event logging systems are the UNIX syslog and Microsoft's event logger.

Logs have been the basis for several FFDA studies. A non-exhaustive list of examples include [4] and [3], where operating system dependability is analyzed using log files, and [7], [1], where supercomputers outages are analyzed starting from different log files available on supercomputing nodes. Despite these efforts, several works have pointed out the inadequacy of event logs for dependability evaluation. Logs can be either incomplete, e.g., non signaled reboots in [3], or misleading, i.e., they can provide ambiguous information [4], [5].

Recent contributions started to address heterogeneity and inefficiency issues of log files. A proposal for a new generation of log files is provided in [8], where several recommendations are introduced to improve log expressiveness by enriching their format. A metric is also proposed to measure information entropy of log files, in order to compare different logging solutions. Another proposal is the IBM Common Event Infrastructure [9], introduced mainly to save the time needed for root cause analysis. It offers a consistent, unified set of APIs and infrastructure for the creation, transmission, persistence and distribution of log events, formatted according to a well defined format.

Other works propose software packages that automate FFDA phases (e.g., data collecting, data coalescing and analysis). An example is represented by MEADEP [10], providing a data preprocessor for converting data in various formats, and a data analyzer for graphical data-presentation and parameter estimation. In [11] a tool for on-line log analysis is presented. It defines a set of rules to model and correlate log events at runtime, leading to a faster recognition of problems. The definition of rules, however, still depends on the log contents and analyst skills.

While these studies represent an important step forward for log-based dependability analysis, they mainly address format heterogeneity issues (i.e., they focus on *what* has to be logged). However, logs incompleteness and ambiguity cannot be solved acting solely on log format. Developers may miss to log particular failure events, and they may produce log events with ambiguous descriptions. At the same time, analysis tools may coalesce uncorrelated events. In this paper logging rules are introduced to define the points in the source code *where* (other than *what*) events should be logged. This allows to improve the detection of failures and related grouping.

III. System model

A system model is necessary to precisely define effective logging rules in a simple and technology-independent way. We aim to propose an *accurate-enough* model, in the sense that system internal components, along with their interactions, can be modeled at a level of detail decided by designers, according to the grain they need, independently from mere technological factors. Following this objective, we classify system components in two categories, according to the following definitions:

Entity: an *active* system component. It provides services that can be invoked by other entities. More in detail, it executes local computations, it starts interactions involving other entities or resources and it can be object of an interaction started by another entity.

Resource: a *passive* system component. At most it is the object of an interaction started by another entity of the system.

The proposed definitions provide general concepts, which can be specialized according to designer's needs. For instance, entities may model processes or threads, i.e., active elaboration components, while resources may model files and/or databases. Furthermore, entities may represent logical components, i.e., all the executable code belonging to a certain library, or package of code, independently of the process or thread executing it.

Entities interact with other system components (i.e., other entities or resources) in order to provide complex services. Even if the most common interaction paradigm is provided by the method invocation, since we are defining a technology-independent system model, we do not focus on a specific real-world interaction mechanism. Nevertheless, we focus on the properties of an interaction: (i) it is always started by an entity (ii) its object can be another entity or a resource (iii) it possibly causes further computations if the object of the interaction is an entity.

Entities and resources can be graphically represented, respectively, as circles and squares, an interaction as a direct edge from the caller entity to the called. After the modeling phase, the target system appears as a direct graph.

IV. Logging rules

Taking into account the proposed model, we define a set of rules suitable for an *external observer* to detect entity failures and to discriminate if they are related to (i) a local computation or (ii) an interaction involving a failed entity or resource. To this aim, each following rule, defines the point of the source code where the event has to be logged.

R1, *Service Start - SST*: the rule forces the SST event to be logged before the first instruction of each service provided by an entity. It provides the evidence that the entity, when invoked, starts serving the requested interaction.

R2, *Service End - SEN*: the rule forces the SEN event to be logged after the last instruction of each service. It provides the evidence that the entity, when invoked, completely serves the requested interaction.

The SST and SEN events are not enough to figure out the cause of an entity failure (i.e., local or interaction). We clarify this concept with an example in the field of the object oriented programming. Let A be an object providing the service named `serviceA()` and `log()` a facility to log the described events. When the method is invoked, the entity (i.e., the object) logs the SST event Fig. 1 (A). If the entity does not log the SEN event, we are not be able to figure out if the outage is due to the local computation (e.g., I1 in Fig. 1 (A)) or to the interaction involving the other entity of the system (e.g., I2 in Fig. 1 (A)).

```
void  A::serviceA(int x){
      log(SST);      //R1
      100/x;         //I1
      b.serviceB(); //I2
      log(SEN);      //R2
}
```

(A)

```
void  A::serviceA(int x){
      log(SST);      //R1
      100/x;         //I1
      log(EIS);      //R3
      b.serviceB(); //I2
      log(EIE);      //R4
      log(SEN);      //R2
}
```

(B)

Fig. 1: Logging rules usage.

For this reason we introduce interaction-related events addressed by the following two rules.

R3, *Entity (Resource) Interaction Start - EIS (RIS)*: the rule forces the EIS (RIS) event to be logged before the invocation of each service. It provides the evidence that the interaction involving the entity (resource) is actually started by the calling entity.

R4, *Entity (Resource) Interaction End - EIE (RIE)*: the rule forces the EIE (RIE) event to be logged after the invocation of each service. It provides the evidence that the interaction involving the entity (resource) ends.

Notice that no other instructions exist between the events EIS (RIS)-EIE (RIE). By using R3 and R4 too, the example code shown in Fig. 1 (A) turns in Fig. 1 (B). In this case, if the interaction `b.serviceB()` fails (e.g., by never ending, as in case of a hang in the called entity), we are able to find it out, since the event EIE is missing.

Typically, an entity offers more than one service or starts more than one interaction. In this case multiple SSTs (EISs) are produced by the entity and we cannot understand to which service (interaction) they are related. To overcome this limitation, the *start* and *end* events related to each service or interaction within the same entity are logged jointly with a unique key. From this point on, each system entity is a black-box: the event flow generated according to the rules, is used by an external observer for the alerts generation process.

V. Events processing

We aim at achieving an understanding of the dependability behavior of the system solely by analyzing the event flow that comes out from each entity. Since logging code interleaves the entity source code, we assume as possible failures of the system entities the ones that result in modification, suspension or termination of the entity control flow thus leading to delayed or missing log events. This assumption indirectly provides possible failure modes. We clarify the concept by examples. The assumption covers crashes or hangs (both active and passive) failures of the system entities. When an entity crashes (or hangs) while serving a certain request, SEN is missing in the related flow. At the same time the calling entity may not be enabled to log the EIE. The assumption, on the other hand, does not fit value failures, but, at the state of art, it is known that they seem to be not detectable solely via logging.

Since our focus is represented by anomalies that result in delayed or missing events we use external detectors based on timeouts. As a reminder log events are provided in *start-end* pairs. A SST has to be followed by the related SEN, an EIS has to be followed by the related EIE (in a similar way for a resource). We measure the time between two related events (i.e., the *start* and *end* events belonging to the same service or interaction) during each fault-free operation of the target system in order to keep constantly updated the expected duration of each pair of events. A proper timeout is then tuned for the anomaly detection process. Fig. 2 clarifies the concept.

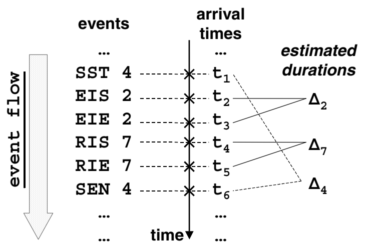


Fig. 2: Event flow analysis.

The external detector generates an alert whenever an *end* event is missing. We define three kind of alerts:

- *entity interaction alert*: it is generated when EIS is not followed by the related EIE within the currently estimated timeout;
- *resource interaction alert*: it is generated when RIS is not followed by the related RIE within the currently estimated timeout;
- *computation alert*: it is generated when SST is not followed by the related SEN within the expected timeout and neither an entity interaction alert nor a resource interaction alert has been generated.

A computation alert represents a problem that is *local* with respect to the entity that generated it. On the other hand, an interaction alert reports a misbehavior due to an *external* cause. The example shown in Fig. 3 clarifies the concept. Let A and B be two system entities. A starts an interaction with B. If a latent fault is triggered during the service execution, B does not log SEN. At the same time A, that is still working, is not enabled to correctly log the EIE because of the B's outage. An *entity interaction alert* and a *computation alert* are raised for A and B respectively.

We generalize the example with respect to the proposed model. A system is made up of components that interact among

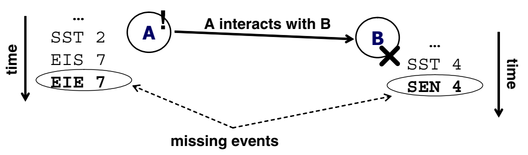
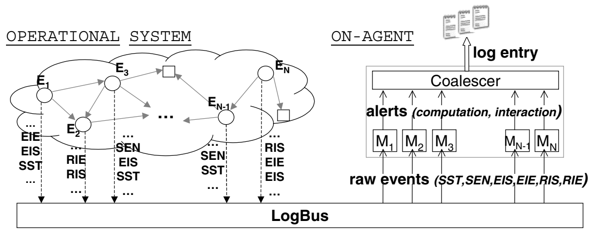


Fig. 3: Example.



OPERATIONAL SYSTEM
ON-AGENT
log entry
Coalescer
alerts (computation, interaction)
raw events (SST,SEN,EIS,EIE,RIS,RIE)
LogBus

Fig. 4: Logging and Analysis Infrastructure.

them in order to provide a certain number of complex services. A long interaction chain is composed by simpler one-to-one interactions between (i) two entities or (ii) an entity and a resource. As shown by the example, if a fault is triggered, we experience a *single* computation or resource interaction alert, related to the component ultimately responsible of the problem, possibly followed by several interaction alerts coming from the entities involved in the interaction chain.

Generated alerts are finally coalesced with the following strategy. When a *computation* or *resource interaction alert* is received, a new tuple is created. Each received *entity interaction alert* is stored until the previous and successive tuple have been created and it is subsequently coalesced with the one that is closer in time. We design a tool (Section VI) that automatically manages the alert generation and coalescing process.

In traditional logging a single triggered fault directly leads to multiple and ambiguous notifications. Analysts have to correlate them to reduce the amount of actually useful information and in order to ideally have a single entry per problem. Figuring out the component that originally caused the problem is a hard task. By using simple and effective rules, we are not only aware of multiple system components that are affected by the same propagating fault but, by means of meaningful alerts, we can also identify with a high level of confidence the triggering component of the outage.

VI. The support tool

In order to ease the production as well as the on-line collection and analysis of events, we design a logging framework partially shown in Fig. 4. It is composed by three key elements: (i) the target system producing events according to the rules (ii) a transport layer named LogBus (iii) a set of *pluggable* components performing different kind of analysis. For the sake of simplicity we are only going to focus on the component implementing the analysis approach described in the previous section. This is named on-agent (ONline-Alerts GENeraTor and coalescer). Fig. 4 gives an overview of the tool internal organization within the context of the cited framework. Further details can be found in [12].

On-agent is organized as follows. Raw events produced by the system entities are preliminary processed by a layer of monitors (i.e., $M_1, M_2, M_3, ..., M_{N-1}, M_N$). These ones generate the alerts with the described technique. A coalescer component receives the alerts and, by analyzing them, produces detailed log-entries permanently stored on files. Each entry contains the timestamp, the name of the system component that originally caused the outage and the names of the affected components.

VII. Experimental results

A. Application

We evaluate the proposed approach and the implemented tools on a real-world case study in the field of Air Traffic Control (ATC). It is a distributed object system depicted in Fig. 5 according to the proposed model.

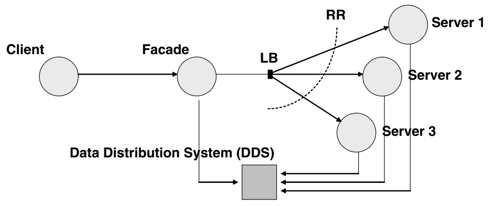


Fig. 5: Experimental testbed.

The Facade object provides methods to insert, to delete and to update a Flight Data Plan (FDP) instance. These services are invoked by the Client object. Once received the request, the Facade object forwards it to one of three Servers selected with a round-robin Load Balancing (LB) policy. FDPs are distributed within the system by using the OMG Data-Distribution Service (DDS) [13]. The architecture also uses some services (e.g., LB) offered by an open-source middleware namely CARDAMOM[1].

We model each object of the system as an entity and the overall DDS as a resource. Interactions among system components are provided by (i) the CORBA-based remote method invocations and (ii) the write/read facilities offered by the DDS API.

We deploy the application on a testbed composed by three hosts (RedHat Linux Enterprise 4, Intel Core 2 2.40GHz equipped) and we instrument it in order to use the proposed logging framework. We also make the Client object to continuously invoke the services offered by the Facade with a frequency of 1 request per second.

[1] http://forge.objectweb.org/projects/cardamom

B. Results

A comparison of the results achievable by traditional logging and the proposed approach is provided. Since we aim at evaluating the effectiveness of each approach, rather than characterizing the behavior of the target application, we do not wait for *natural* occurring errors during the operational time. Instead we deliberately force entity failures (i.e., crashes and hangs in the Facade and in the three Servers) according to the reliability functions shown in Table I (time measured in centiseconds). We find the Weibull distribution a good choice since it has shown to be one of the most used distribution in failure analysis [14]. Any other arbitrary reliability function clearly fits the aim of the test. Different scale parameters assure that Facade and Servers fail with different rates.

TABLE I: Reliability functions

Entity	Function
Facade	$F(t) = e^{0.0000015t^{0.92}}$
Server	$S(t) = e^{0.000005t^{0.92}}$

We instrument the target objects in order to activate faults according to the chosen functions and we leave the system running for 30 days. During this period 3,268 entity failures are generated. We collect entries produced both by the application and by the on-agent tool. We keep separated the two information sources in order to perform the comparison.

The application produces by its own 6,210 alerts permanently stored on files. We use LogFilter described in [1] to reduce the initial number of alerts to exactly 3,268 tuples, i.e., one per each entity failure, by using a time window of 5 s. This choice *overestimates* the results achievable by traditional techniques since the time window is usually selected solely by observing the collected entries without any awareness concerning the faults occurred in the system. Fig. 6 shows the reliability function of the target system, named l(t), estimated by the interarrival times of the collected tuples.

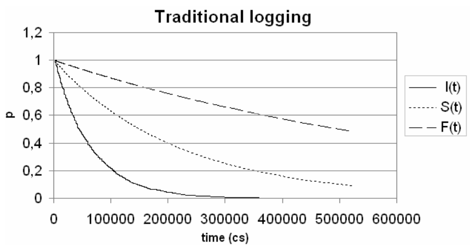


Fig. 6: Reliability function (built-in logging).

The estimated l(t) is different with respect to F(t) and S(t). It does not seem to belong at all to the family of the Weibull functions and a normal distribution may clearly provide a better fit for it. Anyway l(t) represents a correct finding since this is the overall rate that we obtain by letting 4 distinct system entities fail together. The real issue is that, in traditional logging, among multiple notifications, it is difficult to point out the ultimate cause of a faulty behavior of the system, thus making the estimation of the effective failure rate of each component a hard task.

Our approach is intended to overcome these limitations. During the same 30 days period about 40 million of events are sent over the LogBus then automatically reduced to 3,268 effective log-entries, i.e., the number of simulated entity failures, by the on-agent tool. As a reminder, when a fault occurs, several alerts are raised close in time and on-agent produces a detailed log-entry reporting the triggering component of the outage. This makes the categorization of the collected entries *per source* a trivial task. By using the time stamp information of each entry we estimate the fault interarrival time statistical distributions for the Facade object and for each of the three Servers, named f(t) and s(t) respectively.

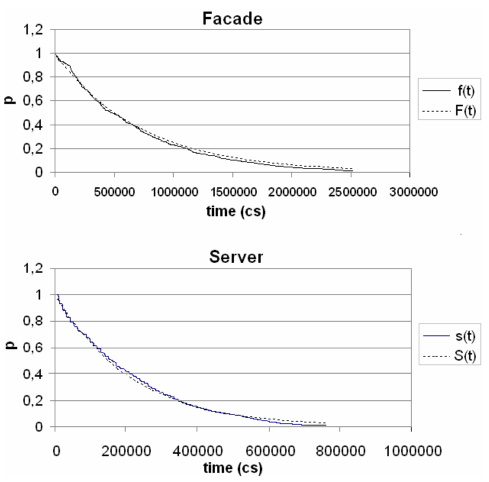


Fig. 7: Reliability functions (proposed approach).

Fig. 7, provides a comparison between the analytical expression of Table I and the related estimated reliability function for each object. We report a single Server due space limitations as similar findings come out for the two remaining ones. Fig. 7 clearly depicts that, by using simple rules and meaningful alerts, we are able to estimate with a high level of confidence the original failure rate of each target component. We perform the Kolmogorov-Smirnov test to prove this statement. Let D be the maximum distance between the analytical and the estimated reliability functions and L be the resulting significance level of the test. Table II reports the obtained results for Facade and Servers. The low value of L assures that the collected samples are consistent with the chosen reliability function.

The estimation of the failure rate is not the only benefit that comes from the use of precise logging rules. As stated, these make possible to understand if a problem with an entity is

TABLE II: Kolmogorov-Smirnov Test

Entity	Samples	D	L
Facade	325	0.0538	L<0.80
Server1	989	0.0378	0.80<L<0.90
Server2	968	0.0357	0.80<L<0.90
Server3	982	0.0338	L<0.80

caused by a propagating fault and thus to prevent erroneous findings. Table III reports the breakup of the total amount of outages for each system entity by local and interaction cause. No local causes are identifiable for the Client as we do not simulate faults in this object. Servers do not exhibit outages due to interaction causes as they do not start interactions with any other object. Finally, the amount of outages for the Facade object is mainly due to interaction causes.

TABLE III: Outages breakup

Entity	Total outages	Local	Interaction
Client	3,268	0	3,268
Facade	3,268	326	2,942
Server 1	990	990	0
Server 2	969	969	0
Server 3	983	983	0

The proposed results show that, analyzing logs by using classical filtering techniques, leads us to observe a system failing with a relatively high rate. With our approach we can conclude that (i) the Facade object is characterized by a failure rate lower than the Servers (ii) being the Server replicated three times its rate is considerably amplified.

VIII. Conclusion and Future Work

The paper proposed a solution to overcome the well known limitations of traditional logging with respect to the dependability evaluation of complex systems. The use of precise rules makes possible to automatically gain information, e.g., the rate and the nature of occurred failures, that turns out to be very useful at improving FFDA results.

Significant improvements are intended to be achieved, including the system model. A layered vision of the system is an essential concern to deal with an in-depth understanding of the nature of failures. We aim at extending the set of proposed rules in order to make possible a coarse-grain characterization of software items (e.g., middleware or operating system) that belong to the executional environment of the target application.

Future work will be also devoted to research for model-driven techniques to automatize the logging-code writing process. This is a need to significantly reduce, if not eliminate, the instrumentation effort.

Acknowledgements

This work has been partially supported by the “Consorzio Interuniversitario Nazionale per l’Informatica” (CINI) and by the Italian Ministry for Education, University, and Research (MIUR) within the frameworks of the “Centro di ricerca sui sistemi Open Source per la applicazioni ed i Servizi MIssion Critical” (COSMIC) Project (www.cosmiclab.it).

References

[1] A. J. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *DSN*, pages 575–584. IEEE Computer Society, 2007.
[2] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *International Conference on Dependable Systems and Networks (DSN 2008)*, Anchorage, Alaska, June 2008.
[3] M. Kalyanakrishnam, Z. Kalbarczyk, and R. K. Iyer. Failure data analysis of a LAN of windows NT based computers. In *Proceedings of the Eighteenth Symposium on Reliable Distributed Systems (18th SRDS’99)*, pages 178–187, Lausanne, Switzerland, October 1999. IEEE Computer Society.
[4] C. Simache and M. Kaâniche. Availability assessment of sunOS/solaris unix systems based on syslogd and wtmpx log files: A case study. In *PRDC*, pages 49–56. IEEE Computer Society, 2005.
[5] M. F. Buckley and D. P. Siewiorek. VAX/VMS event monitoring and analysis. In *FTCS*, pages 414–423, 1995.
[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan.-March 2004.
[7] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. K. Sahoo. Bluegene/L failure analysis and prediction models. In *Proceedings 2006 International Conference on Dependable Systems and Networks (DSN 2006), Performance and Dependability Symposium (PDS)*, pages 425–434, Philadelphia, Pennsylvania, USA, June 2006. IEEE Computer Society.
[8] F. Salfner, S. Tschirpke, and M. Malek. Comprehensive logfiles for autonomic systems. *Proc. of the IEEE Parallel and Distributed Processing Symposium, 2004*, April 2004.
[9] IBM. Common event infrastructure. http://www-01.ibm.com/software/tivoli/features/cei.
[10] D. Tang, M. Hecht, J. Miller, and J. Handal. Meadep: A dependability evaluation tool for engineers. *IEEE Transactions on Reliability*, pages vol. 47, no. 4 (December), pp. 443–450, 1998.
[11] J. P. Rouillard. Real-time log file analysis using the simple event correlator (sec). *IEEE Transactions on Reliability*, page USENIX Systems Administration (LISA XVIII) Conference Proceedings, 2004.
[12] M. Cinque, D. Cotroneo, and A. Pecchia. Towards a framework for field data production and management. In *Proceedings of the International Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems*, pages 34–39, co-located with IEEE SRDS 2008, Naples, Italy, Oct 5, 2008.
[13] Gerardo Pardo-Castellote. OMG data-distribution service: Architectural overview. In *ICDCS Workshops*, pages 200–206. IEEE Computer Society, 2003.
[14] T.-T.Y. Lin and D.P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability*, pages 419–432, 1990.