Run-time fault diagnosis for the Grid

J. Ploski[1], W. Hasselbring[1,2]

[1] OFFIS Institute for Information Technology, 26121 Oldenburg, Germany
[2] Software Engineering Group, University of Oldenburg, 26111 Oldenburg, Germany
email: jan.ploski@offis.de, hasselbring@informatik.uni-oldenburg.de

Abstract

Run-time fault diagnosis means determining the cause of a failure after its occurrence in an operational system. This aspect of systems management consumes significant resources and Grid-based systems are no exception. The paper presents the rationale and the research agenda for a method of diagnosing run-time problems in Grid environments. Components that may contain faults are instrumented to provide data about events. During a first-time diagnosis an expert selects the fault-related events from the observed set. Diagnostic cases are recorded to support reasoning in later, repeated diagnoses. Challenges involved in implementing the proposed method are discussed.

1 Introduction

Run-time fault diagnosis means determining the cause of a failure after its occurrence in an operational system. This aspect of systems management consumes significant resources [1, 2] and Grid-based systems are no exception [3, 4]. For this reason, the stakeholders (developers, administrators and end-users) should take care to optimize their diagnostic processes. We note that the main reason why fault diagnosis is difficult in practice is that due to the considerable size and relative novelty of Grid-based systems the people who perform it must rely on imperfect system models.

While general troubleshooting techniques exist and are known to systems administrators, they are hard to apply without an appropriate system model. For example, techniques which rely on reducing a problem case while comparing the expected and actual test results require not only that a test method is available, but also that the encountered problem is structured and formalized well enough to permit the required step-wise reduction [5]. Because problems in Grid environments tend to lack such a formal structure, the state-of-the-art fault diagnosis relies heavily on what might be called ad-hoc case-based reasoning. When a problem is encountered, the observed symptoms are manually translated into keywords of a query to a general-purpose search engine in hope of finding similar already published problem reports with suggested solutions. When this fails, sources of information such as mailing lists or bug databases are consulted to obtain expert advice. Although this process often works, it is neither precise nor efficient. It depends fundamentally on personal knowledge, with associated risks to the maintainability and sustainability of the deployed systems.

Based on the experience that faults reoccur across time and across system installations, it is reasonable to say that carefully recording fault-related information in order to support future diagnoses has potential benefits. Our research focuses on improving the current forms of describing faults and the associated repairs by adopting a more systematic approach – the event-based run-time fault diagnosis, introduced in this paper.

2 Principles of fault diagnosis

Diagnosis is a form of logical inference in which a probability distribution for a set of hypotheses concerning the presence of hidden, undesirable system states (errors) that require correction (repair) to recover from failure is updated based on symptoms in form of the observable system states or responses to specific interventions. This definition of diagnosis applies regardless of the domain of discourse, although the exact preferred terminology may vary (e.g., patient instead of system, therapy instead of repair in medicine). In our work the term *fault* refers to an error in the system state which is not itself caused by propagation of another error through execution. Thus, the scope of diagnosis is limited to the system consisting of hardware and software and the "why" question does not extend to elements external to the system (such as the users or organizational policies).

The process of diagnosis can employ deductive or inductive methods of reasoning. In deductive diagnosis, fault hypotheses are proved to follow from some system model combined with the information about symptoms. In inductive diagnosis, the truth of fault hypotheses can be assessed by applying the rules of probability theory. In this sense, the process of diagnosis is formal and well-understood. However, probability theory does not prescribe which hypotheses should be evaluated, how strong the logical relationships between the considered propositions are, or how to convert one's informal, experiential knowledge into the initial (prior) probability distribution. In software engineering terms, the probability theory (along with propositional logic) provides a very general meta-model for constructing diagnostic models, with only a few fundamental constraints on their content and manipulation that ensure the consistency, but not the utility of reasoning.

3 An example

As an example of diagnosis in Grid context, consider the following actual scenario related to the widely deployed TORQUE [6] resource manager. A Grid job is submitted to a remote site, in which the Globus [7] middleware invokes TORQUE to delegate the job to a worker node. As it happens, the disk partition on the worker node containing the `spool` directory is full – an error condition which is not directly observable by the job submitter. Therefore, the `pbs_mom` process cannot store the job's output. Consequently, no output is copied to the submitting (head) node after the job's completion, and no output is staged out

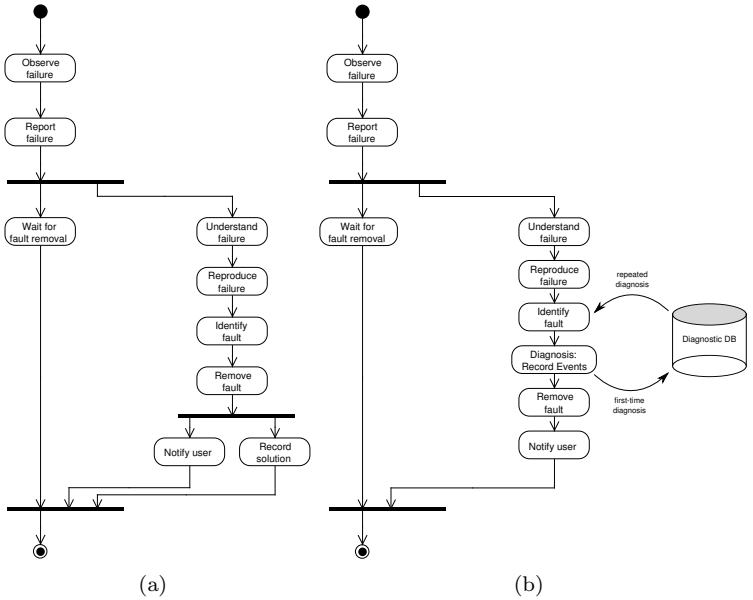

Fig. 1: A traditional procedure for diagnosing faults (a) and its modified version (b) which records the fault-relevant information in form of an event trace to support repeated diagnoses.

by Globus. The user complains to his local administrator about the incorrect behavior of Globus. However, the administrator cannot fully diagnose the problem based on the only visible symptoms. Based on his experience with Globus and knowledge of the system structure, he blames the remote site. He contacts the remote site's administrator and asks him to gather more evidence. At the remote site, multiple fault hypotheses are assumed and falsified one after another to locate the actual fault. After several hours of work during which low-level, generic tools such as `strace` [8] provide data about `pbs_mom`'s behavior, the required repair action is discovered: free up disk space on the worker node. The user is notified when the system's correct service is restored.

The above scenario, outlined abstractly in Figure 1a, is easy to analyze in hindsight. While the few presented inferences are not deductive, they happen at a level approaching certainty. However, deep semantic knowledge about the internal dependencies among the system components is required to enable such inferences. It is not uncommon to "look in the wrong place" for a source of problems (i.e., to mistakenly assign a high prior probability to an unlikely fault hypothesis), and this tendency is aggravated by the unfamiliarity with the diagnosed system. Thus, a similar amount of effort would likely be spent to diagnose the same type of fault again at another Grid site, by another administrator with a comparable amount of experience. A reoccurrence of this scenario does not

seem entirely unlikely; after all, the necessary fault-generating mechanism exists at every Grid site using TORQUE. Finally, the mentioned fault symptoms are difficult to translate into keywords for a text-based search; no distinct quotable error messages were collected in the example scenario. All these observations motivate the need for a more systematic method for preserving fault-related information to support repeated diagnosis.

4 Event-based fault diagnosis

A qualitative examination of diagnostic problems such as the one described in the previous section reveals that they are very context-specific and that their solutions may require an in-depth understanding of the detailed mechanisms that produce the failure. Interactions of different system components, their versions and configurations, the varying availability of system resources, or even misunderstanding and misuse, may all introduce errors that lead to a system failure. Furthermore, sometimes the system behavior classified as failure is not a simple denial of service, but rather a service level degradation, such as when performance decreases or some (but not all) service requests are rejected. In face of this complexity, which is also reflected in research on software fault categorization [10], it is crucial to develop a language which, while more formal and precise than the natural language, still permits modelling realistic problems with an acceptable effort.

We propose *events* as the basic abstraction used to encapsulate diagnostic information (Figure 1b). Events are observable state transitions in system components; they carry some weight in inferences about the system behavior. For example, knowing that the event "a job reached the queue of a local resource manager" has occurred during job processing strongly supports the hypothesis that the job description was syntactically correct (for otherwise the job would have likely been rejected at an earlier stage) and makes virtually certain that the Grid job manager responsible for job delegation was operational (unless another way exists to place a job in the resource manager's queue). In the example from the previous section, knowing that the `write` system call in the `pbs_mom` process failed with return code `ENOSPC` supports the hypothesis that no job output will be staged out (causal, forwards-in-time inference). Likewise, knowing that no output was staged out hints at the possibility of a full worker node disk (diagnostic, backwards-in-time inference). Other interesting relationships among events include specialization (e.g., disk space shortage is a special case of resource exhaustion), composition (e.g., a multi-processor job has finished when all constituent processes have finished), and pairing (e.g., file open/close, lock acquire/release, etc.) The description of a diagnostic case requires a selection of the fault-related events (including the failure events) from among all events that occurred in the system during a fixed time span. A recorded diagnostic case should also contain the recommended repairs that prevent the occurrence of the failure event(s).

4.1 First-time and repeated diagnosis

The concepts of first-time and repeated diagnosis must be distinguished. The former type of diagnosis, conducted by an expert, relies on general diagnostic techniques and on the knowledge about (or a careful examination of) the system internals. In comparison with medicine, this kind of diagnosis can be likened to new drug research, both in character and in cost. However, once the manual diagnosis succeeds, a trace of the representative observable events and their relationships is constructed and recorded to a central diagnostic database for the benefit of a repeated diagnosis of the same fault at another location or time. The repeated diagnosis is comparable to the procedure in a doctor's office. The focus lies on recognizing the symptoms and linking them to the possible hypothesized causes. In software, the data collection can be automated to watch for the known events characteristic to any of the recorded diagnostic cases, thus evaluating an initial set of fault hypotheses which might not be otherwise available to a less experienced diagnoser. Even when no exact match can be made, the diagnostic database can provide valuable information by eliminating unmatched fault hypotheses or providing clues for further exploration by the diagnoser. The retaining, utilization and refinement of diagnostic information in the proposed method differs from the current state of the art, where solutions are either preserved only in the natural language (e.g., through mailing lists or user support system tickets) or not preserved at all (depending on the diligence of the diagnoser).

4.2 Instrumentation

A significant obstacle to the further development of the proposed theory and diagnostic method in current Grid environments is instrumentation. To be able to reason about events, one must first obtain event-based data. Even though data in form of event traces might be more accessible than on-demand data about internal component states, heterogeneous technologies including Java, compiled C code, and scripting languages used in Grids add to the challenge. Some faults exist in even deeper layers of system software (e.g., configuration of OS tools, kernel) that are especially difficult to observe. Instrumentation must deal with the following issues:

- Technical means: most research projects employing instrumentation are limited to software developed using a single, uniform technology. However, despite a trend towards service-orientation and Java, there is little hope that software components implemented in other languages will be eliminated from Grids in the future.
- Efficiency: while instrumentation at low levels of abstraction (e.g., system calls) is often simpler than instrumentation at high levels of abstraction (e.g., application-specific events), the lower-level software generates orders of magnitude more events than higher-level software; if applied at a low level, instrumentation must be smart enough to aggregate event data into meaningful composite events.

- Correlation: events occurring in different components of a software system, at different levels of abstraction, may have to be considered together during diagnosis (e.g., a failed system call in a process can be linked to a job failure). For correlation to be possible, instrumentation must be able to track context information across software layers.

We envision that different instrumentation techniques will have to be applied for different software components constituting a Grid system, united by a common semantic model of events. Another potential direction of research is to obtain event-based data by "log scrapping" – translating already recorded textual information into a format suitable for cross-referencing from diagnostic cases.

4.3 The role of exceptions

Another topic which deserves attention in context of the described method is deciding which kinds of events should be monitored to support the construction of diagnostic cases. The answer is difficult because the system behavior perceived as "failure" can vary greatly according to the user's expectations. For example, in job scheduling at the Grid site level one man's failure is another man's expected behavior. In general, the greater the flexibility of modifying the software's algorithms by its user, the smaller the hope of defining in advance which events are and which are not undesirable.

However, general-purpose programming languages such as Java contain a notion of built-in and user-defined *exceptions* that can be employed at development time to signal detection of an error at run time. In the C language, a similar role is played by *return codes* that, by convention, indicate whether or not an invoked operation succeeded in fulfilling its purpose. Even though the syntactic support for exceptions was introduced to modern programming languages to enable better error handling, empirical studies [11] and programmer interviews reveal that the most common form of "handling" is to log away exceptions, possibly also recovering the system to some consistent state. Thus, exceptions are used in practice to classify events, yet this classification currently only benefits developers during debugging. Based on these observations, exceptions appear as a prime target for monitoring, and the goal of diagnostic instrumentation should be to at least reveal the occurrence of exceptions.

5 Related work

The most closely related work in Grid context is research on Grid monitoring. Monitoring is doubtlessly an important source of diagnostic data. However, past research tends to concentrate on technical aspects (such as the general architecture, performance) and seldom addresses *what* to monitor and *how* to incorporate the monitoring data into troubleshooting. Monitoring techniques can be roughly categorized into passive and active [9], depending on whether the sensors are queried explicitly by a monitoring agent or provide data as a side-effect of the normal execution of processes. Our approach favors passive

techniques, but does not prohibit active checks, as long as they generate events observable through instrumentation. Monitoring of events (rather than just system states) is supported by Grid infrastructure such as R-GMA [12] and event information has been incorporated into simple diagnostic tools in the gLite [13] middleware. Unfortunately, little documentation exists on the kinds and interpretation of events actually captured by these tools; the actual diagnosis, whether first-time or repeated, must be performed by an expert and the tools' scope is limited to one particular Grid middleware.

The topics of (automated) diagnosis and knowledge representation have been studied extensively by researchers in artificial intelligence. Many different approaches have been examined [15], and some of the resulting systems have been successfully fielded in certain application domains [16].

Our own research direction is in particular motivated by Kolodner's case-based reasoning [17], which is a family of approaches used to build knowledge bases for poorly understood and unstructured domains. Software systems are a paradoxal example of such a domain: in principle, they are purposefully designed and can be analyzed by deductive means; most algorithms are intentionally deterministic. However, in practice the effort of analyzing the behavior of large software systems from first principles is so tremendous that alternative models, treating the systems *as if* they were natural objects are appropriate. We must note that the "poorly understood" characteristic of a domain refers to the involved persons more than the domain itself.

Recently, Grids have been slowly entering the field of Enterprise Information Systems, in an effort to spread the technology beyond its origins in scientific high-performance computing. The topic of diagnosis is reflected in this broader field by cmommercially funded research initiatives such as Autonomic Computing [18]. In particular, specialized event-based diagnostic systems are in field use today in network management [19].

6 Conclusions

We have presented the rationale, the core concepts and the research agenda for a new method of fault diagnosis in Grid systems. We hope that this contribution will be useful in broadening the perspective of Grid technologies research and in bringing attention to the more general theory of diagnostic reasoning in troubleshooting computer systems. The successful deployment and long-term operation of Grids will require an increase in their dependability. Reducing the time necessary to identify a fault is one of the means to obtaining such improvement.

Our future work will focus on refining the presented concepts and testing their validity through prototypical implementations.

Acknowledgements. This work is supported by the German Federal Ministry of Education and Research (BMBF) under grant No. 01C5968 as part of the D-Grid initiative.

References

1. Anderson, E. A.: *Researching System Administration*, PhD thesis, University of California, Berkeley, 2002.
2. Barrett, R., Kandogan, E., Maglio, P. P., Haber, E. M., Takayama, L. A., Prabaker, M.: Field studies of computer system administrators: analysis of system management tools and practices. In *CSCW '04: Proc. 2004 ACM conf. on Computer Supported Cooperative Work*, pp. 388–395, New York, NY, USA, 2004. ACM Press.
3. Tierney, B., Gunter, D., McParland, C., Olson, D.: Grid Troubleshooting. In *DOE National Collaboratories Program Investigator's Meeting*, Argonne National Laboratory, IL, USA, August 2004.
4. Medeiros, R., Cirne, W., Brasileiro, F., Sauve, J.: Faults in Grids: Why are they so bad and what can be done about it? In *GRID '03: Proc. 4th Intl. Workshop on Grid Computing*, p. 18, Washington, DC, USA, 2003. IEEE Computer Society.
5. Zeller, A., Hildebrandt, R.: Simplyfying and isolating failure-inducing input. *IEEE Trans. Soft. Eng.*, 28(2):183–200, February 2002.
6. Staples, G.: TORQUE resource manager. In *SC '06: Proc. 2006 ACM/IEEE Conf. on Supercomputing*, p. 8, New York, NY, USA, 2006. ACM.
7. Foster, I. T.: Globus Toolkit Version 4: Software for Service-Oriented Systems. In Jin, H., Reed, A. D., Jiang, W., eds., *IFIP Intl. Conf. on Network and Parallel Computing*, vol. 3779 of *LNCS*, pp. 2–13. Springer, 2006.
8. Sladkey, R., Kranenburg, P., Lankester, B.: strace(1) – trace system calls and signals, Linux manual page.
9. Holub, P., Kuba, M., Matyska, L., Ruda, M.: Grid Infrastructure Monitoring as Reliable Information Service. In Bubak, M., Kacsuk, P., Podhorszki, N., Wismüller, R., Fahringer, T., Malawski, M., eds., *Grid Computing*, vol. 3165 of *LNCS*, pp. 220-229, Berlin/Heidelberg, Germany, 2004. Springer.
10. Ploski, J., Rohr, M., Schwenkenberg, P., Hasselbring, W.: Research Issues in Software Fault Categorization, *SIGSOFT Soft. Eng. Notes*, 32(6):1–8, November 2007.
11. Ploski, J., Hasselbring, W.: Exception handling in an event-driven system. In *Proc. 2nd Intl. Conf. on Availability, Reliability and Security 2007 (ARES '07), Workshop on Event-Based IT Systems (EBITS '07*. IEEE Computer Society Press, 2007.
12. Cooke, A. W. et al.: The Relational Grid Monitoring Architecture: Mediating Information about the Grid. *Journal of Grid Computing*, 2(4):323–339, December 2004.
13. The gLite Middleware, http://glite.web.cern.ch/glite
14. Pearl, J.: *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, Inc., 1st ed., 1988.
15. Reiter, R.: A theory of diagnosis from first principles. In Ginsberg, M. L., ed., *Readings in Non-Monotonic Reasoning*, pp. 352–371. Morgan Kaufamnn, 1987.
16. Deployed Bayesian-Nets Systems in Routine Use, http://www.cs.ualberta.ca/ greiner/BN-Applic/add-new-applic.html.
17. Kolodner, J.: *Case-Based Reasoning*. Morgan Kaufmann, 1993.
18. Kephart, J. O., Chess, D. M.: The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.
19. Steinder, M., Sethi, A. S.: A survey of fault localization techniques in computer networks. *Sci. Comput. Program.* 53(2):165–194, 2004.