Monere: Monitoring of Service Compositions for Failure Diagnosis

Bruno Wassermann* and Wolfgang Emmerich

University College London,
Gower Street, London, WC1E 6BT, UK
{b.wassermann,w.emmerich}@cs.ucl.ac.uk

Abstract. Service-oriented computing has enabled developers to build large, cross-domain service compositions in a more routine manner. These systems inhabit complex, multi-tier operating environments that pose many challenges to their reliable operation. Unanticipated failures at runtime can be time-consuming to diagnose and may propagate across administrative boundaries. It has been argued that measuring readily available data about system operation can significantly increase the failure management capabilities of such systems. We have built an online monitoring system for cross-domain Web service compositions called Monere, which we use in a controlled experiment involving human operators in order to determine the effects of such an approach on diagnosis times for system-level failures. This paper gives an overview of how Monere is able to instrument relevant components across all layers of a service composition and to exploit the structure of BPEL workflows to obtain structural cross-domain dependency graphs. Our experiments reveal a reduction in diagnosis time of more than 20%. However, further analysis reveals this benefit to be dependent on certain conditions, which leads to insights about promising directions for effective support of failure diagnosis in large Web service compositions.

1 Introduction

Service-oriented technologies have simplified the development of larger, more complex software systems that now routinely span administrative and organisational boundaries. These large-scale distributed systems inhabit a complex operating environment that presents numerous threats to their dependability. They are often asynchronous and rely on best-effort networks with variable performance in order to integrate and share resources across domain boundaries. Communication across the Internet and heavy loads increase the potential for failures. Some of the components that are critical to the correct operation of an application may be in different administrative domains. These systems often rely on various layers of middleware components. Developers have been enabled to build such applications as compositions of services in a more routine manner

* This work is partially supported by the EC's 7^{th} Framework Programme under grant agreement n 215605 (RESERVOIR) and a BT EPSRC Case studentship.

through standards and middleware that attempt to hide much of the underlying complexity.

This transparency, which is so useful to developing large service compositions, becomes an obstacle to rapid diagnosis of failures at runtime. Given the complexity of the operating environment and the high demands placed upon it, there may be many unforeseen failures at runtime. Data about failures are spread across the various layers of the system and often across administrative domains. It has been argued [23,27,18,28,9,12] that the availability of large-scale distributed software systems could benefit from improved monitoring capabilities. The definition of availability, *Availability = Mean-time-to-failure / (Mean-time-to-failure + Mean-time-to-repair)*, further supports this intuition. Mean-time-to-repair (MTTR) itself is defined as $MTTR = MTT_{detect} + MTT_{diagnose} + MTT_{repair}$. Approaches to monitoring that improve understanding of system behaviour and thereby enable operators to reduce the time to detect and diagnose failure causes, should result in a significant increase in system availability. However, in practice the benefits of monitoring for the diagnosis of operational failures have only been demonstrated by argument or limited anecdotal evidence.

The contributions of this paper are two-fold. First, it provides a brief architectural and functional overview of the Monere monitoring framework for cross-domain Web service compositions. The idea behind Monere is to support operators with the early identification of system-level failures by measuring readily available attributes of system operation from all relevant components and integrating this data. Monere monitors application-level components, such as BPEL processes and Web services, middleware components including application servers, database servers, BPEL runtimes and Grid computing middleware as well as aspects of the operating system, such as the file system, network interfaces and running processes. We describe how Monere exploits the structure of BPEL workflows and knowledge about other components to create structural cross-domain dependency graphs. Second, this paper presents the results of a controlled experiment, in which we compare the failure diagnosis times achieved by 22 human operators when using Monere to that achieved using a standard UNIX toolset. The results illustrate under what circumstances such an approach is beneficial and point to further work in this area.

We briefly characterise the systems and types of failures we consider based on an example. Then, we provide an overview of the architecture of Monere and its key features, paying particular attention to its mechanism for dependency discovery. This is followed by a performance analysis of Monere, a detailed description of the controlled experiment and a discussion of its results.

2 The Polymorph Search Workflow

Global service compositions exhibit a number of interesting characteristics. They are asynchronous distributed systems communicating over the Internet. As network quality tends to vary and standard communication protocols are used, long delays can become indistinguishable from failed endpoints and lost messages.

A service composition relies on correct service from several layers of often interdependent middleware components. At the service-level, these applications also form complex dependencies across organisational boundaries, where any service may in turn be a high-level application that is composed of more basic functionality.

The Polymorph Search Workflow [14] is our case study. It is a large Web service composition expressed in the Business Process Execution Language (BPEL) [16] and shares many of the above characteristics. It is used by Theoretical Chemists for the computational prediction of organic crystal structures, which is a compute- and data-intensive application. Its middleware consists of several layers. A BPEL runtime is responsible for managing the execution of a set of BPEL processes comprising an application. It is intimately dependent on the services of a SOAP [10] runtime that manages the interactions between Web services. It is also the place where many of the decentralised middleware services, such as transactional mechanisms, are implemented. The SOAP runtime relies on an application server, which is executed by a JVM, and all components use the services of an operating system.

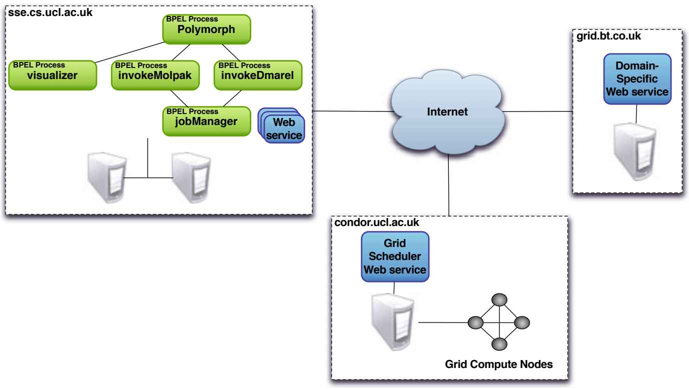

Fig. 1. The Polymorph Search Workflow and its deployment on our testbed

The deployment of the overall workflow on our testbed is shown in Figure 1. The workflow itself is hierarchically composed from several BPEL processes. The processes along with some utility Web services and their backend implementations are deployed across two hosts in the UCL domain. Some Grid computing middleware is used in addition to the standard Web services stack. Access to a set of compute nodes is provided through a Web service and underlying Condor [20] job scheduler instance on a third host. GridSAM [19] manages the interaction between the BPEL processes and Condor. Finally, the workflow also interacts with a Web service deployed in another administrative domain.

The failures we want to enable operators to diagnose more efficiently are operational failures that occur at runtime. System-level failures originate beneath the application-level in the middleware components, operating system and the network. Failure is brought about by sustained high demands on some or all parts of the system, which over time leads to overload and eventually to resource exhaustion. They often depend on the operating environment of the service composition. Examples are the exhaustion of shared resources such as memory, threads or file descriptors by some components, network connectivity issues that prevent timely progress and slowdown of services or components due to being overloaded. Other common causes stem from the cross-domain nature of these applications, where resources and services fail, become unavailable or where varying network conditions make them appear so. Failure effects are almost never isolated or tolerated within a single component or even layer and cascade across layers and even across administrative domain boundaries. These failures often result in abnormal termination or bring the application to a virtual standstill.

3 Monere

Monere implements a number of key features. It automatically discovers all dependencies a BPEL process forms on other application components across domains and lower-level system components within a domain. It continuously collects metrics to aid in understanding of system behaviour and provides historical records of these measurements. The collected lower-level measurements can then be correlated with specific application activity. Finally, Monere integrates all these measurements from the various services, components and hosts in a single user interface.

3.1 Metrics

A great deal of data about system operation can be obtained without the need for substantial changes to the monitored systems. Monere tracks the availability of all discovered components and maintains statistics about their past availability. It listens for error and warning messages from all components that provide log files in the OS, middleware and application. Monere also captures information about the activities performed by the monitored application, such as actions performed by a BPEL process. This affords correlation of application activity with measurements taken at the system level via timestamps.

Performance indicators can be useful in diagnosing the cause of lower than expected performance, reveal network issues or show that a remote service has become overloaded. Examples include request latencies experienced by clients, execution times and throughput of invoked Web services, packet drop rates at the network interface and latencies between hosts. Object-relational mapping tools provide information and various statistics about transactions.

Resource usage metrics facilitate identification of components that have exceeded certain limits, but also deficiencies in the configuration of components, where a demanding workload may result in resource exhaustion. Monere measures OS resources, such as file descriptors, threads, memory and CPU utilisation by load type. It measures the available swap space and the rate at which the OS makes use of the swap. Internal resource usage of Java-based components is obtained using JMX [22] and, as a last resort, Monere integrates metrics that can be obtained from command-line tools.

Measurement collection is driven by collection intervals defined for each metric and current intervals range from five to 60 seconds. Monitoring agents collect measurements through a variety of techniques. For example, Monere subscribes to the BPEL runtime to receive notifications of state changes of monitored processes. Request interception is another common pattern, in which a probe is placed between a client and a server. A third approach is polling, where a variety of means are used to periodically obtain measurements. For example, Monere queries relevant MBeans in JMX servers. The availability of some components can be checked by issuing HTTP HEAD requests or by querying the OS process table.

3.2 Overview

Monere is based on the open-source RHQ enterprise management system [3] and its JBoss extension Jopr [24]. RHQ provides a set of core services for systems management, such as an abstract inventory model and the discovery of hardware and software resources running on hosts based on a simple containment hierarchy. Monere modifies and extends RHQ in such a way as to make it suitable for monitoring cross-domain Web service compositions.

The three main components of Monere are shown in Figure 2. Each domain hosts a central Monere server, which communicates with the deployed agents in its domain. The server obtains resource discovery results and measurements from its agents, persists this in a database and then makes this data available to end users via a user interface. The agents, which are deployed on the hosts they monitor, register with their Monere server and then periodically examine their environment to discover resources and obtain measurements. A Monere

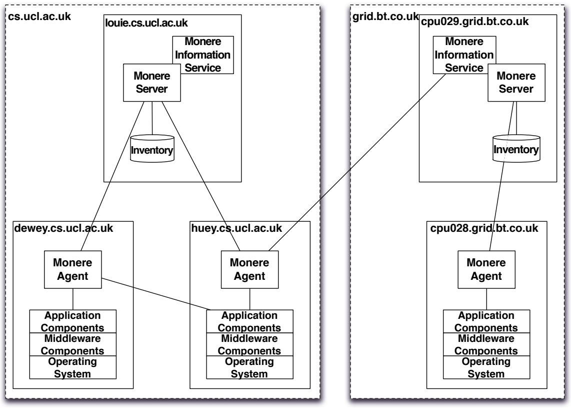

Fig. 2. The key components of Monere in a typical deployment

agent is essentially a runtime system for a set of plugins and manages communication with the Monere server. Plugins represent particular resource types and encapsulate the functionality for their discovery and the measurement of corresponding metrics. Plugins consist of a descriptive part expressed in XML and a Java implementation for the process of discovery and measurement collection. Examples of resources represented by plugins include a Tomcat server, an Axis SOAP runtime, the Web services deployed within it and the various components of a Linux OS.

The Monere Information Service (MIS) addresses the requirement of providing visibility of dependencies on resources in remote administrative domains and sharing information about their state. Service providers can make certain metrics about their published services available to clients. These metrics currently include the throughput and execution times of Web services. An agent that has discovered a dependency from one of its local resources onto a Web service in another administrative domain, can subscribe to an RSS feed at the corresponding MIS. The MIS restricts visibility to the level of published Web services and provides no further insight about the underlying infrastructure. It thus safeguards the sensitivity of implementation details, while providing data that can help to determine problems with remote services.

The Monere user interface has been built using Adobe Flex [1]. The UI is composed of a number of panels, each of which provides a particular view onto the available monitoring data. The Resource Tree view (Figure 3) provides an hierarchical view of all discovered components and the Dependency view displays the dependents and dependencies of a selected component along with basic information, such as its current availability. There are panels to display the activities performed by BPEL processes and report on any severe log messages from any part of the monitored system. The UI provides more detailed information on request. The Selected Resource Metrics view displays the metrics of any resource dragged onto it. The values are updated in real-time and data is aggregated. Charts of historical records help to identify trends and anomalies.

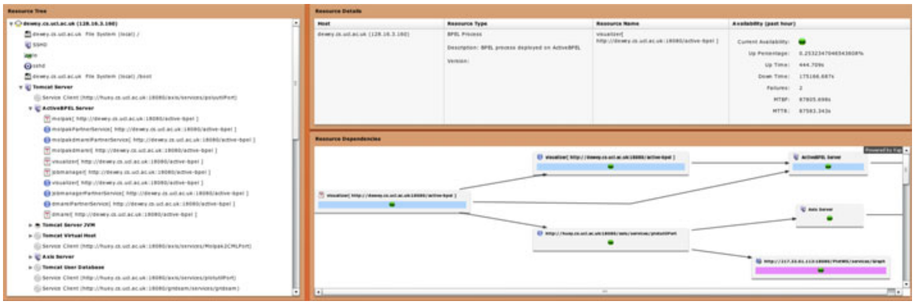

Fig. 3. Panels of the Monere UI showing dependencies of a selected resource across hosts and administrative domains

3.3 Dependency Discovery

Our dependency graph represents all relevant dependencies from the viewpoint of an application. This includes dependencies among application-level components, such as BPEL processes and Web services, across host and domain boundaries and dependencies on and among the lower-level components down to the operating system. Conceptually, Monere's dependency discovery process can be described in three phases. The first phase is concerned with the discovery of local components by each agent. An agent iterates over its deployed plugins and invokes their discovery interface implementations. Each plugin knows how to discover a corresponding component on the local host and returns a representation of it along with relevant attributes. We use a number of different mechanisms for discovery. Hyperic's SIGAR API [2], implements native libraries for a number of operating systems and can discover OS components, such as running processes, file systems and network interfaces. Another mechanism is to query JMX servers for specific MBeans that represent components of interest or parse deployment descriptors in the case of application servers. Some components offer administrative interfaces, which allow for programmatic queries on their properties and deployed components.

For each type of component, the RHQ plugin XML definition specifies what other types of components it is hosted by. For example, a BPEL process 'runs-in' a BPEL runtime. This information enables an agent to establish simple containment relationships between components. Another type of knowledge about dependencies is encapsulated within the implementation of some of the discovery code where, upon discovery of a component, an agent is instructed to parse specific configuration files to find dependencies on other middleware components. If two components are on the same host, the agent immediately establishes the dependency. Otherwise, the agent will insert a placeholder component into its inventory. This placeholder contains sufficient information about the remote component for the monitoring server to resolve it to the actual component discovered by another agent.

In the second phase, agents carry out a static analysis of the discovered application components on their hosts. BPEL processes explicitly identify the partner Web service interfaces with which they interact. Furthermore, WSDL documents import other WSDL service interfaces that they depend on. An agent begins by parsing the discovered BPEL processes to obtain information about the Web service interfaces used as partner services. The corresponding WSDL binding definitions are then parsed to link this abstract information about interfaces to the URL of the corresponding Web service instance. The process continues by parsing the documents of the identified interfaces and resolves further dependencies recursively. When an agent discovers a dependency on a Web service on another host or in a different administrative domain, it can insert a placeholder as the agent on the remote host is responsible for the corresponding part of the dependency graph.

In the final phase of the process, the agents submit their local inventories to the monitoring server. Upon submission by an agent, the server processes the simple containment hierarchy and high-level dependencies among application-level components. It replaces any placeholder components with dependencies on the corresponding components from other hosts already committed to its inventory. This approach captures structural dependencies as opposed to functional ones obtained dynamically. In practice, the discovered dependencies represent functional ones well. However, a number of relationships are omitted, such as links to the backend implementations of Web services or the file system.

4 Performance Analysis

In order to determine the **impact on application performance**, we have executed 30 runs on the Polymorph service composition with and without monitoring enabled on real input data sets. We find a relative slowdown of about 8%, or a mean runtime of 199.23 seconds compared to 181.63 seconds. The slowdown is not high given the number of different components and metrics, but it is not negligible either. Our measurements of the **per-host overhead** imposed by monitoring agents over a 60 minute period are summarised in table 1. The agents monitor about 340 metrics each and, given the current set of collection intervals, each take about 500 to 600 measurements per minute. The number of components and metrics in this case is relatively large as is the variety of the measurement techniques used. As can be seen in table 1 the requirements on the CPU[1] and memory are low.

Table 1. Performance overhead of two monitoring agents

Ag	Components	Metrics	CPU	Heap Usage
1	66	341	3.2%	11.3MB range:5.7
2	90	349	4.2%	18.1MB range:6.9

Analysing **local communication overhead**, we can assume that there will always be new measurements to report on within the 30-second report interval, given that many of the metrics are continuous. Approximately, the size of measurement report (MR) data within an administrative domain is given by

$$a \times \prod_{i=0}^{n} \frac{30}{c_i} \times |metrics_i| \times k, \tag{1}$$

where a is the number of agents in an administrative domain, c_i is a particular collection interval, $metrics_i$ is the number of metrics at this collection interval and k is some constant for the size of a measurement. The smallest collection interval Monere supports is one second, which imposes an upper bound on the fraction of 30. As the collection intervals increase, we see that the cumulative size of exchanged MRs is given by the product of the number of agents and the number of metrics, with the latter one likely to be the dominant factor.

[1] A single core Pentium 4 3.2 GHz.

Measurements performed with *tcpdump* over a period of 30 minutes reveal a bit rate within an administrative domain of 3.06Kbits/s and 0.48Kbits/s for the exchange between administrative domains.

5 Experiment

The controlled experiment examines the performance of 22 human operators tasked with the diagnosis of a number of typical system-level failures in order to quantify the effects of Monere on diagnosis times.

5.1 Experiment Setup

We have replicated the Polymorph workflow and its infrastructure on a testbed with two separate domains, as shown in Figures 1 and 2. Each participant was asked to diagnose six failures injected into the system. The failures were injected in random order and participants were given 10 minutes per failure. The stated goal was to determine the root component(s) of the failure and identify what about its behaviour is likely to have caused the observed problem. Participants were randomly assigned to use either the Monere prototype or a standard set of tools. Both groups were given a brief written overview of the Polymorph service composition and the toolset to be used. The control group's tools consisted of root shells onto the hosts in the local domain, JConsoles connected to the local Tomcat JVMs, pointers to log files and a brief explanation of Condor shell commands. Control group participants furthermore had access to a session of the ActiveBPEL monitoring console [4], which provides an overview of the state of executing BPEL processes. Monere participants had only access to Monere's web-based UI.

Each group consisted of 11 participants. The participants were volunteers from among the post-docs and faculty of several CS departments and also included a small number of professional software engineers. The level of experience of each participant was determined through a questionnaire. The levels of experience between the groups are well-balanced, but we omit a closer analysis of this aspect for space reasons.

The failures were selected among a larger set of failures as they were observed with the Polymorph composition running in production. Injection was through a script-driven framework we have developed to reliably reproduce the conditions that lead to the six failures. These failures are:

1. unavailability of a remote service without known cause
2. slowdown of a remote service caused by overload
3. application server failure caused by thread exhaustion
4. application server failure caused by heap exhaustion
5. unavailability of the grid scheduler due to disk space exhaustion
6. failure to schedule compute jobs in a timely manner due to overload

5.2 Results

Success Rate. We examine the proportion of successfully diagnosed failure causes and find that Monere participants achieve a success rate of 95% on average, while the Control group only diagnosed 72% of the presented failures correctly. For the Control group, the two most frequently misdiagnosed failures are the ones affecting the Web service hosted in the remote administrative domain. As such, this is not a surprising result. However, it does serve to demonstrate the usefulness of cross-domain dependency graphs, which enable operators to more quickly identify when an issue lies outside the local infrastructure. The most frequently misdiagnosed failure for the Monere group was failure 5. This highlights a weakness of structural dependency graphs, where some of the existing functional dependencies (i.e. between Condor and the file system) may not be represented explicitly.

Diagnosis Time. Next, we examine whether there is a statistically significant difference in the observed diagnosis times. We refer to the mean times to diagnose the six failures as MTT$_{diag}$.

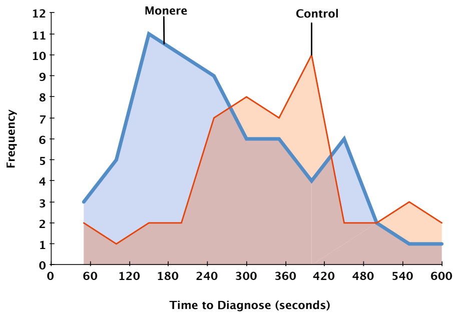

Fig. 4. Histogram of the diagnosis times for the Monere and the Control group

Table 2. Summary of testing for a significant reduction in mean diagnosis times

	MTT_{diag}	StdDev	n	t_0	$t_{0.05,109}$
Monere	241.55s	132.34	63	-2.80	1.659
Control	311.96s	129.71	48		

Figure 4 is a histogram of the diagnosis times of each group and table 2 summarizes our observations. The histogram suggests that operators using Monere achieve shorter diagnosis times, but there is also a cluster of observations that reveals a higher number of almost seven-minute diagnosis times. The MTT$_{diag}$ for the Monere group is 241.55 seconds. The participants in the Control group diagnosed the same failures with a MTT$_{diag}$ of 311.96 seconds. We perform a one-sided t-test at the 95% confidence level in order to determine whether the diagnosis times achieved by Monere are indeed shorter with statistical significance.

For this, we need to confirm $t_0 \leq t_{0.05,109}$. The degrees of freedom are obtained from the number of observations $(63 + 48 - 2 = 109)$. As the resulting values show, $-2.80 < 1.659$. This allows us to reject the null hypothesis in favour of the alternative hypothesis, which states that participants using Monere do indeed achieve shorter diagnosis times. We find an average reduction in MTT$_{diag}$ with Monere of 22.5%.

Failures. We were are able to identify some failures with noticeably shorter and longer median diagnosis times than the rest. For the Control group, the failure with the highest median diagnosis time is failure 1 (remote service unavailable) and the lowest times are achieved for failure 5 (Condor unavailable). As table 3 illustrates, these failures are exactly reversed for the Monere group.

Table 3. Failures with longest/shortest median diagnosis times

	Control	Monere
Longest	Failure 1 (419s)	Failure 5 (430s)
Shortest	Failure 5 (264s)	Failure 1 (76s)

The relatively long diagnosis times for failure 1 are due to lack of tool support that provides adequate visibility of critical components in separate administrative domains. This forces Control group participants to examine many components in the local infrastructure before focussing their attention on the remote dependencies. Control group participants identified failure 5 relatively quickly as upon issuing one of the available Condor commands, they would immediately learn that the Condor process was down, which prompted them to inspect its log file and based on that examine the available disk space. Monere participants had much more difficulty identifying this problem as the available dependency graph did only reveal a dependency between Condor and the OS. This left participants with a larger problem space to explore. Conversely, for failure 1 the dependency graph enabled participants to quickly identify that they were dealing with an issue outside their local domain.

5.3 A Short Cost-Benefit Analysis

So when is the investment in online monitoring actually justified? To answer this question, a service provider would need to determine two things. First, one needs to have an accurate idea of the downtime cost per unit of time for a particular service offering. For cross-domain systems, this information should be available at the time of writing an SLA in the form of financial penalties associated with the violation of promised service level agreements [25]. Second, it is necessary to estimate the typical proportion of diagnosis time on overall downtime for a service implementation and its system-level failures. This information could be obtained either through an organisational baseline or, once available, from detailed measurement-based failure studies on large service compositions.

In general, we can conclude that an approach such as Monere will only be worth it when applied to systems whose overall failure repair times are clearly dominated by their diagnosis times and where the financial penalties or reputational loss for performing below agreed service levels are substantial.

5.4 Validity

Ideally, we would have liked to install Monere on an application in production and compare the performance of system operators to an organisational baseline. As this is not feasible outside an industrial setting, we rely on a single service composition and limit the number of failures to six. This limits the generalizability of our results to some extent. However, the Polymorph composition is a real-world application of moderate size and complexity and has been executed under realistic workloads. It encompasses most of the challenging characteristics we are interested in. Furthermore, the selected failures are not trivial to diagnose. The internal validity is strong as each participant uses only one of the tool sets and is presented with each failure only once and in random order. The measures used (i.e. success rate, diagnosis time) are objective ones. The number of data points on diagnosis times are sufficient, even though they could be improved on. The failure injection method we employ does not merely simulate the symptoms of a failure, but reproduces the actual conditions in the system that lead to it.

6 Related Work

6.1 Dependency Discovery

Indirect approaches to dependency discovery trace some aspect of system operation and perform statistical analysis in order to determine likely dependencies. In [15] readily available performance data, such as request counts and times, are measured in order to to determine containment of activity periods of transactions between pairs of nodes. The frequency of such containments between the same pairs of nodes gives an indication of the strength of their relationship. The rate of false dependencies increases with the rate of concurrency in the system. The addition of a statistical model to estimate false positives, [17] improves the accuracy of dynamic dependency graphs. Their approach needs to intercept messages that pass network devices. In Pinpoint [13] middleware services within a J2EE server are instrumented to trace the traversal of requests and learn likely call paths.

The main advantages of indirect approaches are that they discover functional dependencies and rely primarily on instrumentation at the middleware and network layer. Monere discovers structural dependencies. The resulting dependency graph may contain relationships that become active only rarely. Nevertheless, these structural dependencies seem to correspond well to functional ones and do not require a large number of observations under varying workloads. Even though some manual effort is required for the expression of model information, this knowledge is expressed modularly (i.e. per component type) and can be reused among many deployments. Finally, Monere does not rely on any instrumentation beyond what is already made available by the OS and discovered components.

6.2 Monitoring

Monere differs from much work on SOC monitoring ([6], [21]) in that it does not primarily focus on the composition- and service-level and associated higher-level events. Instead, it regards these as one of several types of components and events to be integrated from all relevant layers of a middlware-based distributed system. Magpie [7] obtains control paths and resource usage for the threads involved in servicing requests by instrumenting the Windows NT kernel. It records events at points where control flow transfers among components and uses request schemas to associate events with observed requests. vPath [26] is another approach, which does not require propagation of request identifiers. It instruments virtual machine monitors to capture thread activity and TCP system calls. Assuming synchronous request-reply invocations and a dispatch-worker thread model, it can link recorded activity and requests.

These approaches are quite elegant in that they exploit observable events that occur in response to end-user requests to infer resulting system activity. Monere provides an evidence-based determination of the impact of online monitoring at all layers of a service composition on diagnosis times and shows under what circumstances this is actually justified.

Several approaches focus on tracing events at the kernel-level. Chopstix [8] instruments the operating system kernel to collect measurements about process scheduling, I/O activity, system calls and socket communication. Two other approaches that avoid instrumentation above the level of the operating systems are DTrace [11] and SysProf [5]. DTrace enables instrumentation of user-level and kernel-level events through user-specified probes. SysProf instruments the kernel to record resource consumption by capturing context switches and system calls.

Observing events at the kernel-level provides visibility of everything that happens in a system, requires only a small number of measurement techniques and affords instrumentation of legacy applications. A downside is that operators are left to figure out how kernel-level events relate to particular application-level activities. Monere demands considerable development effort given the variety of required measurement techniques. Its advantages are that it produces more familiar metrics, makes it easier to understand how parts of the system react to particular application activity and should be easier for a broader range of developers to extend.

7 Conclusions

We have presented the results and insights from an empirical evaluation of the effect of monitoring readily available data about system operation on mean diagnosis times for system-level failures in large distributed systems. We have developed a prototype of such an approach called Monere and given an overview of its architecture and measurement techniques. We have also shown how it is possible to obtain dependency graphs through modular models and by exploiting the structure inherent to BPEL workflows. A brief analysis reveals that such an approach will outweigh its performance cost in cases where diagnosis time is the dominant factor of overall system downtime and where the financial penalties associated with the violation of SLAs are substantial. Our results also provide some insights about the utility of dependency graphs and how missing or inaccurate relationships can be so misleading during diagnosis as to reduce an operator's performance to below that of someone not having access to such a graph.

One key lesson from our investigation is that the collection of a wide array of measurements about system operation is not in itself sufficient to drastically reduce diagnosis times in most scenarios. Measuring aspects of system operation is important in order to keep applications operational. However, availability of rich repositories of data, rather than being the solution are a prerequisite for providing higher-value added functionality. In our view, a promising direction to improve the handling of unanticipated system-level failures in large Web service compositions is to investigate automated mechanisms that can determine anomalies and other areas of interest within a large volume of data and thereby effectively reduce the problem space on behalf of system operators.

References

1. Adobe Flex, `http://bit.ly/2DbkE9`
2. Hyperic SIGAR API, `http://bit.ly/96BIG3`
3. RHQ, `http://bit.ly/apijCR`
4. ActiveBPEL (2010), `http://bit.ly/be87LF`
5. Agarwala, S., Schwan, K.: Sysprof: Online distributed behavior diagnosis through fine-grain system monitoring. In: ICDCS (July 2006)
6. Baresi, L., Guinea, S.: Self-supervising bpel processes. IEEE TSE 37, 247–263 (2011)
7. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using magpie for request extraction and workload modelling. In: OSDI. USENIX, Berkeley (2004)
8. Bhatia, S., Kumar, A., Fiuczynski, M.E., Peterson, L.: Lightweight, high-resolution monitoring for troubleshooting production systems. In: OSDI. USENIX, Berkeley (2008)
9. Birman, K., van Renesse, R., Vogels, W.: Adding high availability and autonomic behavior to web services. In: ICSE. IEEE CS, Washington, DC, USA (2004)
10. Box, D., et al.: Simple Object Access Protocol (SOAP 1.1) (May 2000)
11. Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: ATC. USENIX, Berkeley (2004)
12. Chandra, A., Prinja, R., Jain, S., Zhang, Z.: Co-designing the failure analysis and monitoring of large-scale systems. SIGMETRICS Perform. Eval. Rev. 36 (August 2008)
13. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: problem determination in large, dynamic internet services. In: DSN. IEEE (2002)

14. Emmerich, W., Butchart, B., Chen, L., Wassermann, B., Price, S.L.: Grid Service Orchestration using the Business Process Execution Language (BPEL). JOGC 3(3-4), 283–304 (2005)
15. Gupta, M., Neogi, A., Agarwal, M.K., Kar, G.: Discovering Dynamic Dependencies in Enterprise Environments for Problem Determination. In: Brunner, M., Keller, A. (eds.) DSOM 2003. LNCS, vol. 2867, pp. 125–166. Springer, Heidelberg (2003)
16. Jordan, D., et al.: Web Services Business Process Execution Language 2.0 WS-BPEL (August 2006)
17. Kashima, H., Tsumura, T., Ide, T., Nogayama, T., Hirade, R., Etoh, H., Fukuda, T.: Network-based problem detection for distributed systems. In: ICDE. IEEE CS, Washington, DC, USA (2005)
18. Katchabaw, M., Howard, S., Lutfiyya, H., Marshall, A., Bauer, M.: Making distributed applications manageable through instrumentation. In: Proc., 2nd Intl. Workshop on SEPDS 1997 (May 1997)
19. Lee, W., McGough, S., Newhouse, S., Darlington, J.: A standard based approach to job submission through web services. In: Cox, S. (ed.) Proc. of the UK e-Science All Hands Meeting, Nottingham, pp. 901–905. UK EPSRC (2004) ISBN 1-904425-21-6
20. Litzkow, M., Livny, M., Mutka, M.: Condor - A Hunter of Idle Workstations. In: ICDCS (June 1988)
21. Moser, O., Rosenberg, F., Dustdar, S.: Event Driven Monitoring for Service Composition Infrastructures. In: Chen, L., Triantafillou, P., Suel, T. (eds.) WISE 2010. LNCS, vol. 6488, pp. 38–51. Springer, Heidelberg (2010)
22. Perry, J.S.: Java Management Extensions, 1st edn. O'Reilly & Associates, Inc., Sebastopol (2002)
23. Plattner, B.: Real-time execution monitoring. IEEE TSE 10(6), 756–764 (1984)
24. Red Hat, Inc.: Jopr. `http://www.jboss.org/jopr`
25. Skene, J., Raimondi, F., Emmerich, W.: Service-level agreements for electronic services. IEEE TSE 36(2), 288–304 (2010)
26. Tak, B.C., Tang, C., Zhang, C., Govindan, S., Urgaonkar, B., Chang, R.N.: vpath: precise discovery of request processing paths from black-box observations of thread and network activities. In: ATC. USENIX, Berkeley (2009)
27. Vogels, W.: World wide failures. In: Proc. of the 7th Workshop on ACM SIGOPS European Workshop: Systems Support for Worldwide Applications. ACM, New York (1996)
28. Vogels, W., Re, C.: Ws-membership - failure management in a web-services world. In: WWW (2003)