Data-Driven Detection and Diagnosis of System-Level Failures in Middleware-Based Service Compositions

Bruno Wassermann

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

University College London.

Department of Computer Science

University College London

September 2012

I, Bruno Wassermann, confirm that the work presented in this dissertation is my own. Where information has been derived from other sources, I confirm that this has been indicated in the text.

Abstract

Service-oriented technologies have simplified the development of large, complex software systems that span administrative boundaries. Developers have been enabled to build applications as compositions of services through middleware that hides much of the underlying complexity. The resulting applications inhabit complex, multi-tier operating environments that pose many challenges to their reliable operation and often lead to failures at runtime. Two key aspects of the time to repair a failure are the time to its detection and to the diagnosis of its cause. The prevalent approach to detection and diagnosis is primarily based on ad-hoc monitoring as well as operator experience and intuition. This is inefficient and leads to decreased availability.

We propose an approach to *data-driven detection and diagnosis* in order to decrease the repair time of failures in middleware-based service compositions. Data-driven diagnosis supports system operators with information about the operation and structure of a service composition. We discuss how middleware-based service compositions can be monitored in a comprehensive, yet non-intrusive manner and present a process to discover system structure by processing deployment information that is commonly reified in such systems. We perform a controlled experiment that compares the performance of 22 participants using either a standard or the data-driven approach to diagnose several failures injected into a real-world service composition. We find that system operators using the latter approach are able to achieve significantly higher success rates and lower diagnosis times.

Data-driven detection is based on the automation of failure detection through applying an outlier detection technique to multi-variate monitoring data. We evaluate the effectiveness of one-class classification for this purpose and determine a simple approach to select subsets of metrics that afford highly accurate failure detection.

Acknowledgements

I would like to thank my supervisor, Wolfgang Emmerich, who has allowed me considerable freedom in performing this research, while also helping me to stay focussed on bringing it to completion. His style of supervision and his experience, which he so generously shared with me, has provided me with a tremendously valuable learning experience on many levels, both professionally and personally. Thank you.

I would also like to thank Licia Capra for her detailed reviews and suggestions about earlier incarnations of my work as a reviewer in both my first and second year vivas.

Brad Karp's honest feedback has taught me more about my research in an evening than I could have hoped to discover myself in an entire year. I am grateful to him for articulating clearly what was only present in a latent form in my mind.

I thank my examiners, Aad van Moorsel and Emmanuel Letier, for taking the time to first work through this dissertation and then for allowing me to discuss it with them in detail.

I am grateful to Franco Raimondi, Clovis Chapman, Rami Bahsoon and Mirco Musolesi for proof-reading this dissertation at short notice.

I am also grateful for the support provided by my sponsor, British Telecom. Mike Fisher and Paul McKee offered interesting insights about the practicalities of managing large-scale systems and generously contributed part of the computational infrastructure that was used in our experiemental evalution.

I would also like to thank the members of the Software Systems Engineering group and of the department of Computer Science at large. I have had the pleasure of meeting too many interesting people to name here individually; their companionship over the last couple of years has been invaluable.

Finally, I would like to thank my parents and my wife, Nathalie, for supporting me throughout this entire endeavour simply because it was important to me and in spite of the sacrifices it may have demanded at times. Thank you.

I dedicate this dissertation to my late father, Moses Wassermann, of blessed memory, to my mother, Hanna Wassermann and to my wife and daughter Nathalie and Noa Wassermann.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Amazon Web services (AWS) is a large Cloud computing provider that offers on-demand computing facilities. It supports the development of applications on top of its platform through a set of services that manage access to computational resources, storage, databases, and so on. A large number of businesses rely on the infrastructure and on the services provided by AWS in order to offer their own services to a worldwide audience of customers across the Internet. AWS has suffered from several wide-ranging outages. In April 2008 [AWS, 2008], an internal networking issue resulted in many of its clients' services becoming unavailable for about 90 minutes. AWS clients include popular Internet services such as reddit [red, 2012] and foursquare [fou, 2012]. These services depend on the reliable operation of the infrastructure services offered to them by AWS. Each failure in the infrastructure of AWS has the potential to quickly propagate its effects and affect the services of its clients. In this particular instance, the system operators at AWS expended 96% of the overall time to resolve this issue on diagnosing its cause. In 2010 [Fac, 2010], a failure caused the Facebook API to become unavailable, which affected more than 300,000 Web sites using this API. It took system operators two and a half hours to identify the cause and fix it. Engineers at Google were still investigating the root cause of an outage to GMail after it had first been noticed by users [goo, 2009]. These are just a few examples.

The Software-as-a-Service (SaaS) and Infrastructure-as-a-Service (IaaS) paradigms and corresponding service-oriented technologies have simplified the development of complex distributed software systems that routinely span organisational boundaries. Middleware facilitates the development of complex applications as compositions of basic services. It makes higher-level abstractions available to developers. These abstractions encapsulate expertise about various aspects of distribution and relieves developers to focus increasingly on application logic instead of low-level issues. By attempting to hide much of the underlying complexity that arises from the integration of widely distributed and heterogeneous resources, middleware enables a wide range of developers to build cross-domain, middleware-based service compositions.

The resulting applications are large-scale distributed systems and often inhabit a complex, multi-tier operating environment, which presents numerous threats to their reliable operation. These systems often make use of asynchronous and wide-area communication. A high sustained workload may accelerate

the effects of resource leaks and therefore increases the potential for failures. Such service compositions typically rely on the service of several layers of middleware components. Some of these components encapsulate substantial functionality and expertise about different aspects of the operation of a service composition. Furthermore, these components are inter-dependent to some extent and form a network of dependencies on each other. Cross-domain service compositions integrate resources across administrative boundaries which further limits control and visibility of components that are nevertheless critically important to the reliable operation of an application. The resulting service compositions are subject to frequent failures.

We will have an opportunity to characterise the types of failures that we consider in this thesis and that we refer to as *system-level failures* in the next chapter. In short, system-level failures have severe consequences on an application, causing either its abnormal termination or severely degraded performance and prevent an application from making progress, unless the failure is detected and its cause is identified and repaired. System-level failures originate in the middleware, the operating system or the network and have a negative impact on the availability of service compositions.

How can we increase the availability of large service compositions? One definition of availability [Candea, 2004] is as follows: *Availability = Mean-time-to-failure / (Mean-time-to-failure + Mean-time-to-repair)*, MTTF (the average amount of time between successive failures) needs to be large in relation to MTTR (the average time to repair a failure) in order to increase availability. There are two general ways to increase the MTTF; by removing the faults that give rise to failures or by applying techniques that enable a system to tolerate their occurrence at runtime. Increasing the MTTF in middleware-based, cross-domain service compositions is difficult for a number of reasons. We cannot assume access to source code or that the developers of a service composition possess the required expertise to competently test and debug the numerous components that a service composition depends on. The providers of these middleware components can be assumed to have performed extensive testing and debugging prior to release. Furthermore, the lack of control and in some cases even visibility of components that are hosted in different administrative domains is another factor that makes it difficult to approach the problem by trying to increase the MTTF. It is safe to assume that most large Internet services apply state-of-the-art fault removal and fault tolerance techniques and processes. Nevertheless, their service offerings are still subject to frequent, wide-ranging and prominent outages. As we will demonstrate in the next chapter, system-level failures are neither easily removed prior to deployment nor tolerated after it and consequently often need to be handled at runtime in order to reduce their negative impact on the availability of service compositions.

If our best efforts at fault removal and fault tolerance leave residual faults and vulnerabilities to severe failures, then another approach to increase availability is to decrease the time to repair failures (MTTR). The idea to improve the availability of software systems by targeting the MTTR of failures originally formed the foundation of Recovery Oriented Computing efforts at Berkeley University [Patterson et al., 2002]. In order to decrease the MTTR, we need to be able to handle the occurrence of failures at runtime more efficiently. That is, we need to improve the failure management capabilities

of service compositions, which is based on the three activities of detection, diagnosis and the subsequent repair of a failure. These activities also comprise the MTTR, which can be defined as $MTTR = MTT_{detect} + MTT_{diagnose} + MTT_{repair}$. In this thesis, we focus on the first two of these activities (detection and diagnosis) for two reasons. First, the successful detection and diagnosis of a failure often precedes the ability to effect appropriate repair actions. And second, as we explain in the next chapter, detection and diagnosis of system-level failures tends to be difficult.

The standard process to detect system-level failures in middleware-based service compositions is, to a large extent, a manual one. It is based on a system operator continuously observing the various components that comprise a service composition in order to spot behaviour that is indicative of failures and to do so before users suffer from the consequences of a failure and complain about it. This process to failure detection is time-consuming, slow and error-prone. The currently prevalent approach to diagnosis primarily relies on the experience and intuition of a system operator. The data needed to test her hypotheses about the possible causes for a given failure are spread throughout the system and may not be available anymore after the fact as there is often a lack of detailed historic records beyond the information that is available from some log files. This can result in a process of trial-and-error that necessitates attempts to reproduce a failure and capture the conditions that may have led to it by temporarily installing more verbose monitoring. This cycle often needs to be repeated, if the assumptions about a possible failure cause are not confirmed. The cross-domain distribution of resources causes a lack of visibility and control over some components, which further complicate the detection of failures and the identification of their causes.

We propose an approach to the *data-driven detection and diagnosis* of system-level failures in middleware-based cross-domain service compositions. Middleware-based service compositions produce a variety of data as a by-product of their operation. While this data is scattered across layers, hosts and administrative domains, it is possible to collect much of it in a non-intrusive manner. This allows us to integrate it from its various sources, correlate it with application activity and maintain historic records of measurements. These measurements capture metrics that describe various aspects of the behaviour of a service composition, such as its availability, application state changes, performance and resource usage. Furthermore, middleware and service compositions reify a great deal of information about the deployment and calling relationships that exist between the components and services in a given deployment. We can process this information in order to automatically discover all components that comprise a service composition along with the dependencies they form among each other within and across hosts as well as across administrative domain boundaries at the service level.

The data-driven approach to diagnosis is supported by both the comprehensive, non-intrusive monitoring of a service composition and by the discovery of structural, cross-domain dependency graphs. System operators are provided with access to both. They can inspect historic and real-time measurements about the behaviour of a service composition and can correlate events that describe application activity with these measurements. Furthermore, the dependency graphs provide an overview of the structure of a given service composition and can guide system operators in their search through the large

problem space they are faced with in a more efficient manner. This offers an alternative to the traditional diagnosis process, which is primarily based on ad-hoc measurements and the experience and intuition of a system operator.

The data-driven approach to detection is based on applying classification-based machine learning techniques to a large repository of measurements about system operation as it is provided by our monitoring approach. This affords the automated detection of system-level failures in a timely and highly accurate manner.

This thesis presents the building blocks on which the data-driven approach is based and presents an evidence-based determination of its effects.

1.2 Bounding Assumptions

Our approach caters for middleware-based service compositions. We do not assume that the identified issues of failure-proneness and inefficient failure detection and diagnosis exist to a similar extent in middleware-less distributed systems, such as for example the Domain Name System (DNS) [Mockapetric, 1987]. In our view, middleware-based and middleware-less or lean distributed systems differ from each other in a number of ways. First, a middleware-based application usually implements more expansive business logic than a lean system that focusses on a specific piece of functionality. Therefore, the developer of a lean distributed system may have more time to cater for issues of distribution and reliability. Second, middleware-based systems are expected to conform to standard interfaces and provide common quality of service guarantees in standard ways (i.e. certain message delivery semantics and mechanisms) in order to support interoperability. Lean distributed systems can impose their own interfaces on clients and are free to design custom protocols in order to improve certain quality of service concerns. Finally, as developers of middleware-based systems rely on middleware to handle many of the complexities that arise from distribution, some may be less familiar with the intricacies of network programming than is the case for programmers of lean distributed systems that forego the use of such assistance.

We address the detection and diagnosis of a type of failure that we refer to as system-level failures. These failures usually result in the abnormal termination of an application or its severe performance degradation. We omit from consideration the following kinds of failures. Input-dependent failures that are activated by certain points in the input space of an application (e.g., input parameters that cause never-ending computation). We assume that these need to be handled by a domain expert. Design and application logic errors are generally addressed by a large body of approaches on software testing.

Automated localization of failures, even though it is a closely related problem, is left for future work. The work in this thesis forms some of the necessary foundations for such future work as we explain in the final chapter.

1.3 Main Hypothesis and Contributions

The main thesis which we examine in this dissertation can be articulated as follows.

By making use of data about system operation and system structure that can be obtained in a non-

intrusive manner from most middleware-based service compositions, it is possible to support a data-driven approach to the detection and diagnosis of system-level failures in cross-domain, middleware-based service compositions that significantly improves the performance of these two activities.

We will define the corresponding hypotheses more formally in terms of independent and outcome variables in the evaluation chapters (Chapters 5 and 7). At a high level, our expectations are as follows. We expect that data-driven diagnosis, based on the integration of a varied set of metrics about system operation and the discovery of information about system structure, all of which can be obtained without necessitating intrusive changes, can support system operators to engage in a more principled diagnosis process and thereby enable them to perform more successful diagnoses within a shorter amount of time than is possible with the currently prevalent approach, which relies more heavily on the experience and intuition of a system operator. Furthermore, we expect that it is possible to achieve highly accurate and timely failure detection in an automated manner based on the availability of historic records of data about system operation.

We examine this thesis through the following four contributions.

1.3.1 Comprehensive Non-Intrusive Monitoring

We describe how to monitor a service composition at all its layers (i.e. service level, middleware, operating system) and across administrative domain boundaries in a non-intrusive manner in order to integrate and correlate data about system operation that would otherwise remain scattered and hidden. The collected monitoring data forms a repository of measurements which can be used by system operators during diagnosis and by automated mechanisms for failure detection.

Our work on a monitoring framework prototype provides us with a few insights. First, it demonstrates what measurement approaches are available to monitor middleware-based systems without requiring intrusive changes to any of its components. Second, we learn what types of metrics can be measured in this way. And finally, we are able to quantify the performance overhead of monitoring in this manner in terms of the resource requirements of such a monitoring framework, the slowdown of the application that is being monitored and the communication overhead that is incurred from the exchange of measurements within and across administrative domains. We find the incurred overheads are sufficiently low so as to be tolerable by modern networks and servers.

Our monitoring framework does not propose any new monitoring techniques. However, it is the first framework to monitor a service composition at all its layers in a non-intrusive manner that has been described in the literature. Furthermore, the data about system operation that results from this kind of monitoring forms one of the foundations of the data-driven approach. As such, our work on the monitoring framework allows us to evaluate the feasibility of our proposed approach by demonstrating that data about various aspects of system operation can indeed be measured non-intrusively and by allowing us to quantify the performance overhead such an approach imposes.

1.3.2 Cross-Domain Structural Dependency Discovery

We describe a process for the automated discovery of structural, cross-domain dependency graphs. The process exploits various sources of deployment information as they exist in middleware and information

about calling relationships contained in service composition artefacts. The discovery process is bootstrapped by modular per-component models as we discuss in Chapter 4. The graphs capture most of the components that comprise a service composition and several types of dependencies among them within and across hosts and across administrative domain boundaries at the service level. The process avoids intrusive instrumentation by relying on readily available sources of information.

Our work on dependency discovery allows us to determine the accuracy and completeness of structural dependency graphs that are based on information as it is commonly reified by middleware-based service compositions. Similarly, we are able to characterise the limitations of such an approach. Finally, we briefly discuss how the model of dependencies that these graphs provide clients with, facilitate the exchange of measurements between service providers and their clients across administrative domains.

The resulting dependency graphs represent information about the structure of a given service composition that can be made available to system operators during diagnosis. As such, it represents the second foundation of the data-driven approach to diagnosis. Our approach is an extension of previous work by [Kar et al., 2000], which proposed to exploit information contained in operating system software package managers to discover the dependencies on a host. We build on this by integrating various additional sources of information as they are commonly available in middleware-based service compositions.

1.3.3 Data-Driven Failure Diagnosis

We report on the results and insights from an evidence-based determination of the effects of data-driven diagnosis on system operator performance. For this purpose we have performed a controlled experiment with 22 human participants who were asked to diagnose several system-level failures that we injected into a real-world, cross-domain service composition. We compared the performance of participants who used a traditional approach based on a standard tool set with that of participants who engaged in data-driven diagnosis with the help of a prototype that provided them with access to monitoring data and dependency graphs.

Our results contribute new knowledge in several ways. We find out whether data-driven diagnosis can indeed support system operators to achieve significantly higher success rates and lower diagnosis times. Our experiments reveal that participants who use data-driven diagnosis are enabled to achieve significantly higher success rates than those participants using a standard approach. Furthermore, we were able to confirm our expectation that data-driven diagnosis can support system operators to achieve significantly reduced diagnosis times. We confirm a reduction in excess of 50% for failures whose cause originates in remote administrative domains and a reduction in excess of 20% across all examined failures. Our results have enabled us to determine the circumstances under which the application of data-driven diagnosis is most beneficial and justified. We explain how this depends on several factors, such as the cost of downtime as agreed in service-level agreements, the proportion of repair time that is generally expended on diagnosis and the degree to which an application integrates resources that are widely distributed. Fourth, we confirm that structural dependency graphs can provide effective guidance to system operators with navigation of the large problem space that system-level failures confront them

with. And finally, we discover how inaccurate or missing relationships in dependency graphs can have a drastically negative impact on the performance of a system operator.

Our efforts represent the first evidence-based determination of the effects that such a data-driven approach to failure diagnosis has on the performance of system operators. That is, we quantify the effects of making data about system operation and structure available to system operators. Another aspect of the novelty of this contribution is that it argues for the provision of dependency graphs as an important feature to support system operators with identifying failure causes in large systems more efficiently.

1.3.4 Data-Driven Failure Detection

We report on the results of our evaluation of data-driven detection. We have performed a controlled experiment in which we train and then test a recently proposed one-class classification algorithm [Hempstalk et al., 2008] on close to 100 minutes of monitoring data to determine how accurately we can detect failures by applying outlier detection to multi-variate monitoring data.

We gain several insights from this evaluation. We experiment with several strategies to fragment the large space that is created by a varied set of monitoring data in order to select subsets of metrics that can afford the training of highly accurate classifiers. From this, we learn that fragmentation approaches based on human intuition and expertise do not necessarily work well and find an automated approach that does. This approach ranks attributes by their suitability for generating accurate decision trees and then creates differently-sized subsets ordered by the ranking of the attributes. This approach results in five classifiers that afford highly accurate failure detection. Each of these classifiers detects more than 90% of failures and their corresponding false alarm rates range from just one false alarm every seven hours to 11 per hour. Small subsets of attributes (two, three or four attributes) are sufficient to train highly accurate classifiers and generally smaller numbers of attributes are well-correlated with higher levels of accuracy.

There are three novel aspects to our work on automated failure detection. First, we evaluate the suitability of a one-class classification algorithm [Hempstalk et al., 2008] that has not been evaluated for failure detection in distributed software systems so far. Second, we automate failure detection based on a large set of different metrics about system operation, whereas many related approaches use small sets of often hand-picked metrics. And finally, in contrast to related approaches that do use a larger number of metrics, we evaluate several strategies for selecting subsets of metrics in a manner that support accurate failure detection.

1.4 Relevant Publications

Several of my publications are related to the work described in this thesis. They are by no means required reading in order to understand this thesis, but rather capture the evolution of my understanding of this research in a few snapshots.

Our latest paper at the time of writing this thesis is [Wassermann and Emmerich, 2011], in which we report on the initial results of our controlled experiment on data-driven diagnosis. We have since obtained more data points and have analysed them in several ways, which has led to a more nuanced set

of insights.

[Wassermann, 2010] is an ICSE doctoral symposium paper, in which we propose to support system operators through failure monitoring and briefly present an idea to enable the systematic testing for vulnerabilities to system-level failures based on the replay of monitoring traces.

The work described in [Wassermann et al., 2009] was performed during an internship at IBM Research. The project dealt with configuration and change management in cross-organisational environments. My contribution consisted of the design and development of a decentralised change management process that enables different parties to cooperate in the implementation and testing of changes to infrastructure that represents common dependencies in a coordinated manner, operates on an Internet scale (more or less) and respects the autonomy of the various parties. This is the source of the mechanism described in Section 4.5 for the exchange of certain measurements across domain boundaries by allowing clients to subscribe to notifications of updates to their remote dependencies.

[Wassermann and Emmerich, 2006] is an early workshop paper from before I began my PhD studies. We identify the reliability challenges that middleware-based cross-domain service compositions face. We also describe an ambitious set of goals that include enabling non-expert users to perform failure management for what we referred to as global scientific BPEL workflows at the time. The paper expresses an interest in finding out how to divide responsibility for failure management between autonomic mechanisms and human beings.

Finally, in [Emmerich et al., 2005] we describe an early incarnation of the Polymorph Search workflow. One of its later versions serves as the case study used in our evaluation.

1.5 Dissertation Outline

In Chapter 2 we motivate the research problem and our approach based on an example service composition, which we use to characterise the systems and types of failures we consider in more detail. We review some existing approaches to support the reliable operation of distributed systems to demonstrate that they do not adequately address the identified failures and provide a brief overview of our data-driven approach. The comparative literature reviews are presented in chapters 3, 4 and 6.

We continue by first presenting a conceptual overview of the foundations of the data-driven approach so that we can understand what we then evaluate in the experiments. The foundations of data-driven diagnosis consist of the provision of monitoring data about system operation and of dependency graphs that convey system structure.

In Chapter 3 we present a conceptual overview of our approach to non-intrusive monitoring and explain how it can support a data-driven approach to detection and diagnosis. We review closely related work on monitoring.

In Chapter 4 we describe a model of our structural cross-domain dependency graphs and the process used to automate their discovery by exploiting information that is reified by middleware and service compositions. We review existing approaches to the discovery of the structure of distributed systems post their deployment.

With this background having been established, we can proceed to present our evaluation. We begin

with our experiments on data-driven diagnosis first, as its results demonstrate that our failure injection reproduces the conditions reliably and that these conditions become visible in the available monitoring data, given that all failures are diagnosed by some of the subjects.

In Chapter 5 we present the results of our evaluation of data-driven diagnosis in the form of a performance analysis of the underlying non-intrusive monitoring approach, a qualitative review of the accuracy of the resulting dependency graphs and the results of our controlled experiment to reveal the effects of data-driven diagnosis. We begin with a brief overview of our prototype tool support to allow system operators to engage in data-driven diagnosis. Together, Chapters 3, 4 and 5 represent our work on data-driven diagnosis.

Before presenting our evaluation of data-driven detection, in Chapter 6 we first provide the necessary background about the proposed approach to automate failure detection and outline one of its key challenges, which is to cope with the large feature space that is created by multi-variate monitoring data. We compare our efforts with a number of related approaches that detect failures by processing some data about system operation.

In Chapter 7 we present the results of our evaluation of data-driven detection. We discuss a set of controlled experiments which examine several fragmentation strategies used to select effective metric subsets and that quantify the level of accuracy with which we can automate failure detection. The pre-requisites for this chapter are the overview of the multi-variate monitoring data that can be obtained in a non-intrusive manner as described in Chapter 3 and the background about the application of classification-based machine learning to this data in Chapter 6.

Finally, in Chapter 8 we summarise our contributions in terms of what we have learned and discuss a few of the remaining open questions. We close with suggestions for how to build on our efforts.

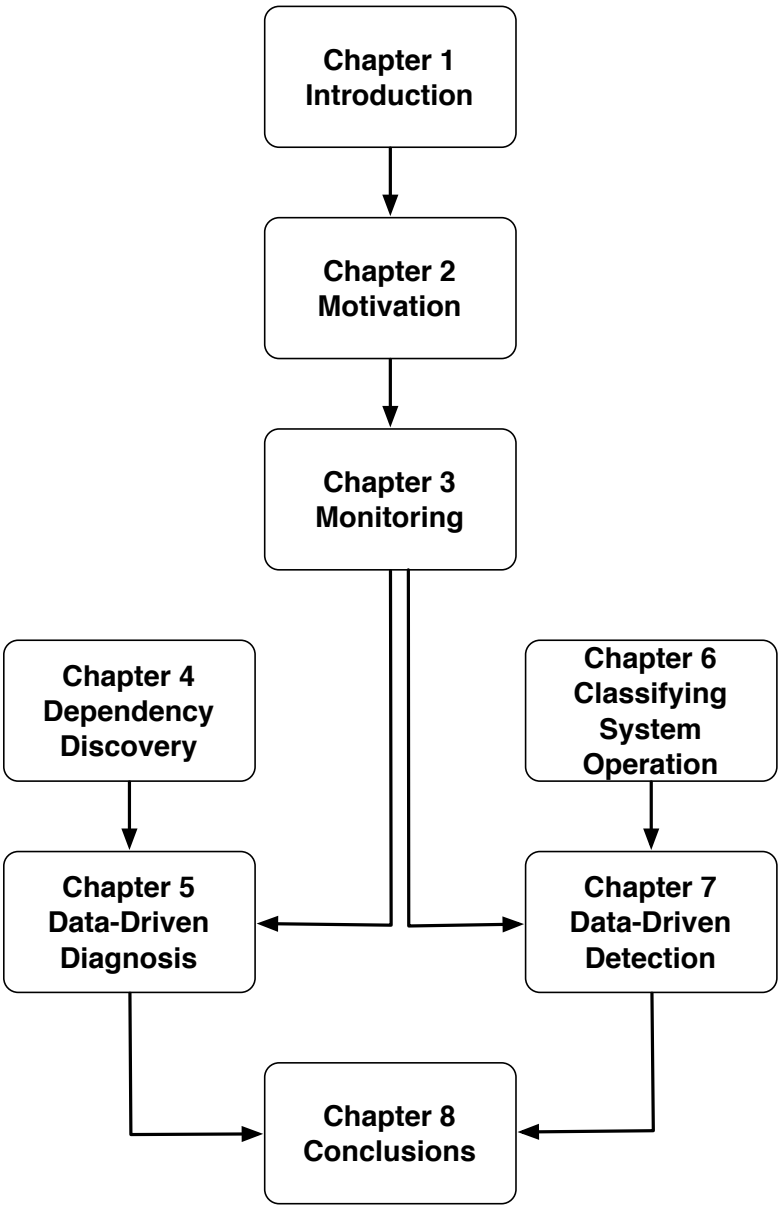


Figure 1.1: Outline of the chapter structure.

Chapter 2

Motivation - The Need for Failure Management

In this chapter we further motivate the research problem and our approach. We do so based on an example service composition, which allows us to characterise the systems and the types of failures we consider in this thesis. We identify factors that make the detection and diagnosis of failures a particularly challenging issue in these systems. We review existing approaches to ensure the reliable operation of distributed systems and point out their limitations in addressing the identified problems. We then close this chapter with an overview of our approach.

2.1 Motivating Example

Our example application is a Web 2.0 news aggregator called Propaganda. A large number of different news sources has become available on the Internet in the form of established news outlets as well as independent ones and niche blogs. As a consequence, readers can encounter conflicting reports presenting different viewpoints about a news story. Many find this unsettling as it reveals the bias inherent in much news reporting and forces them to invest effort in order to learn about the facts behind a story. The Web 2.0 news aggregator addresses this problem. Instead of confronting news subscribers with potentially conflicting information, it aggregates and presents only news stories that are fully consistent with a subscriber's worldview. The aggregator creates a detailed profile of a subscriber's worldview by analysing her activites and connections on a well-known social network and then selects stories from a variety of sources that conform to this worldview. Users access Propaganda through a web-based user interface, via which, after having signed up for the service and having authorised Propaganda to access data from their social network services, they can log in to consume the selected news stories. In this way Propaganda can save subscribers from thinking critically about the news.

Propaganda has been built by a lean Web 2.0 startup that does not have considerable computational infrastructure of its own. Instead, it realises its service offering through the composition of services offered by a number of different providers as shown in Figure 2.1. Propaganda's frontend and integration logic runs within virtual machines on the infrastructure of a compute service provider, which offers on-demand access to computational resources such as units of CPU, memory, persistent storage space

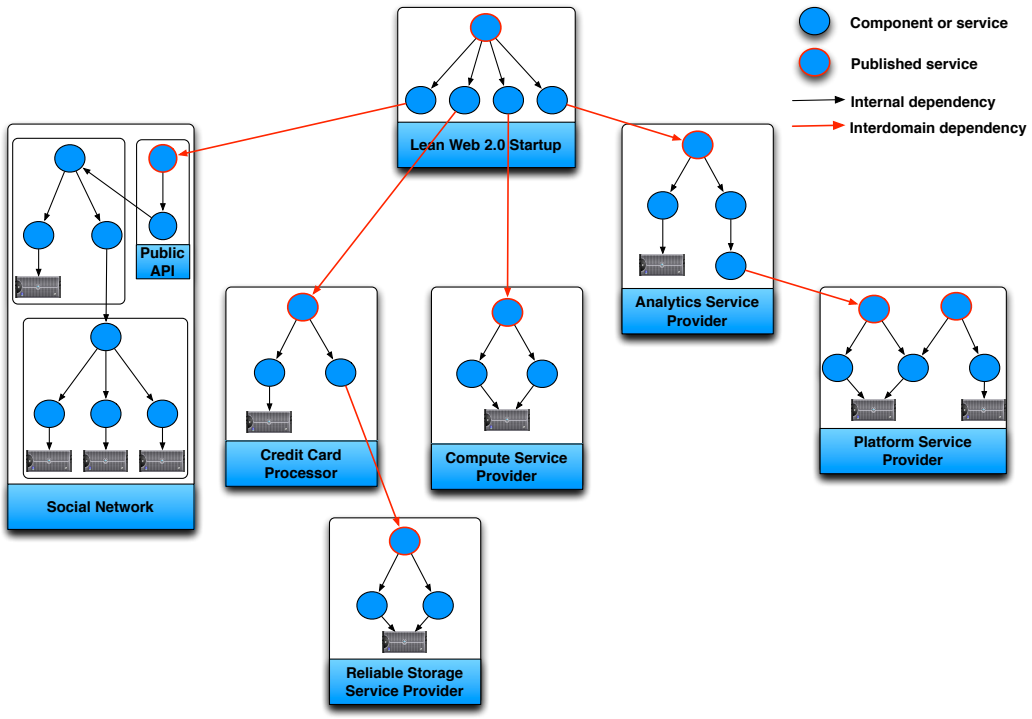


Figure 2.1: A high-level overview of Propaganda, a Web 2.0 news aggregator application developed as a service composition.

and network bandwidth. Access to these resources is negotiated and managed through a Web service interfaces at the compute service provider. Propaganda handles billing of its clients by using the services of a credit card processor. An important requirement on the credit card processor is to maintain secure and reliable records of the processed credit card transactions. As this would be prohibitively expensive to develop in-house and given that malfunction bears potentially considerable liabilities, the credit card processor in turn relies on the services of a storage service provider. Its clients store and retrieve data via published Web services. These provide access to an underlying implementation that is certified to be secure and reliable to required levels.

Propaganda continuously gathers various kinds of data about its subscribers' activities within the social network service. The social network provides access to this data through a Web service that exposes a public Application Programming Interface (API). This API, among other things, enables third parties to retrieve certain types of data about its users with their permission. The data is then formatted for submission to an analytics service. The analytics service encapsulates extensive expertise in applying data mining techniques to the online behaviour of a user in order to derive detailed socio-demographic profiles.

The founders of the analytics service are academics with expertise in data mining of social networks, but they lack experience with large-scale system administration of the sort necessary to maintain a high-volume Web service. Therefore, they rely on a platform service provider to host the application on their behalf. After deploying their application, the platform service provider adjusts its scalability within specified limits in response to concurrent client requests.

This example illustrates how a non-trivial application can be developed as a composition of existing services instead of implementing all components in-house. [Papazoglou, 2008, p. 5] defines a *Web service* as a "*self-describing, self-contained software module available via a network, such as the Internet, which completes tasks, solves problems, or conducts transactions on behalf of a user or application.*" The functionality encapsulated by a Web service can range in granularity from just a simple method to a complex application consisting of several subsystems. When we use the word "service" in this thesis, we either refer to a Web service as just defined or to the type of features, functionality or value delivered by a software component.

Three primary paradigms for service offerings have emerged in industry. In *Software-as-a-Service* (SaaS) offerings, the service is an application developed and hosted by a provider. This can either be an interactive application, such as a spreadsheet editor, word processor or email client or it can be a service whose interface is invoked programmatically. The storage service provider, credit card processor and analytics service are examples of SaaS. Payment consists of a monthly fee or is based on the number of invocations made to the service. The compute service provider is an example of an *Infrastructure-as-a-Service* (IaaS) offering. In this scenario, Web services provide access to computational resources, such as CPU, memory, disk space and network bandwidth. Clients deploy operating systems and entire software stacks on this infrastructure to host applications. In IaaS scenarios payment is based on the amount of computational resources used. Finally, *Platform-as-a-Service* (PaaS) providers offer a platform that is optimised for particular types of applications (e.g., interactive Python web applications). Clients develop an application on top of a standard software stack, consisting of the necessary application servers, databases and other middleware commonly used by such applications. Upon deployment of the application the PaaS provider then manages the entire software stack in order to maintain desired service levels. Payment is usually based on a notion of application size and the degree of concurrency it will handle. The case study we use in our evaluation does not use elements of IaaS and PaaS, but we mention these types of offerings here for completeness.

Building software systems as compositions of services offers several advantages. First, the identification of suitable services replaces, at least to some extent, the engineering of the subsystems required to implement an application. Furthermore, in SaaS the application or service is kept up-to-date by the provider and is available for integration into one's own application across the Internet. The IaaS model takes advantage of the economies of scale achievable by operators of large data centres to offer computational resources at low prices and to enable customers to pay only for those resources that they actually need. Making use of PaaS offerings allows customers to focus on their application logic and leave the administration of the common underlying software stack (e.g., web servers, databases) to the platform provider. Web services facilitate the development of non-trivial applications.

Applications built as service compositions tend to form networks of dependencies as the high-level overview in Figure 2.1 illustrates. We identify two types of dependencies. *Intra-domain* dependencies are contained within a single administrative domain. They represent dependencies on and among the components in the local infrastructure that support the service level. Examples of such components are

application servers, database management systems and distributed caches. Intra-domain dependencies can be considerably more complex than is suggested in Figure 2.1 as we will discuss in more detail. A second type of dependency is between services in separate administrative domains. We refer to these as *cross-domain* dependencies. Cross-domain dependencies can cross organisational boundaries or exist within a single organisation. For example, the social network in our example is a large, widely distributed application. Over time many subsystems have been developed by different organisational units using a variety of different programming languages. These subsystems have been exposed as services based on a common interface definition language and interaction protocol in order to improve the integration of the various subsystems among different administrative domains.

The existence of such dependencies has an impact on the reliability of a service composition. When software systems were under the control of a single administrative domain, most of the critically important software components were known. This made it possible to identify potential threats to the reliability of an application and protect against these. However, a service composition depends on an array of services offered by different organisations. As such it becomes dependent on their underlying implementations, which are usually hidden behind published service interfaces. In addition, cross-domain dependencies can be transitive. For example, Propaganda depends on the analytics service, which in turn relies on the platform service provider. Any outage in the platform provider's implementation has the potential to cascade across these chains of dependencies and negatively impact the service Propaganda offers to end users. The reliable operation of a service composition thus requires correct service from components it has neither control nor even visibility of.

2.1.1 Middleware

Service compositions are essentially wide-area, middleware-based distributed systems. Services may differ in their interface definition languages and underlying implementation technologies, but most share a set of common functional and non-functional requirements that are supported by middleware. A review of some of the tasks that the implementation of the credit card processor service and of the social network service have to perform in order to process requests serves to demonstrate the need for middleware.

The credit card processor accepts incoming requests to process transactions via a web server. Once the web server receives a request, it processes it to determine its intended recipient, which in this example is the service runtime hosting the processor Web service. The service runtime extracts the invoker's message from the request payload in order to determine which service and operation should be invoked. In this instance, the payload is an encrypted XML representation of the pending transaction containing credit card details, the payment amount and information identifying the vendor. The service runtime relies on another component for decryption, which implements a standard Web service specification for securing message contents between two service endpoints. Once the payload has been decrypted, the service runtime employs an XML processor in order to unmarshall the message contents from XML into a programming language object representation. This object is then used in the invocation to a stub representing the service implementation.

A *stub* encapsulates the functionality needed to invoke a remote interface as a local procedure.

Based on an interface definition, which specifies the interface of a server process in terms of procedures and their signatures, an interface compiler can generate client and server stubs. The client and server processes can then communicate with each other via their local stubs. A stub implements the location of and binding to a suitable server process. It is also responsible for converting data from and into common data formats in order to hide any differences in data representation between the two processes.

In this case, the service is actually implemented as a workflow. This workflow encapsulates the various activities and rules involved in processing a credit card transaction. Upon invocation of the corresponding stub in the service runtime, a new instance of this workflow is created and executed within a workflow engine, which in turn depends on an application server. The workflow interacts with a number of Web services at different banks in order to request account debit and credit operations. In order to maintain consistent state at all sites in light of failures, the workflow engine relies on a transaction coordinator component that is deployed within the same application server.

The details of each credit card transaction and its outcome are submitted to the reliable storage service provider. The submission needs to succeed even in the presence of communication failures or temporary unavailability of the storage service. An activity in the workflow creates a data structure containing details about the completed transaction and an XML processor is used to convert this data structure into its XML representation. The workflow then invokes another local Web service, which is responsible for submitting this data to the storage provider. This service relies on the local service runtime in order to invoke the storage provider service on this message. The service runtime again encrypts the message contents and packages it within an HTTP request message. It then hands this message over to a message queuing system that stores the message persistently and guarantees eventual delivery to the storage service endpoint. Finally, the workflow engine creates a response message, encoded in XML using the XML processor and passes the message to the service runtime, which encrypts it and packages it as an HTTP response that the Web server delivers to the client.

The implementation of the social network service illustrates a few additional use cases for middleware. The public API service allows clients to retrieve data about users of the network. When one of its web servers receives a request, it is first passed to a request router, which selects one of several request processors in order to share the load efficiently. Each request processor is an implementation instance of the public API service. It runs within an application server, which manages access to operating system resources on its behalf and provides access to other middleware services. For example, each request is processed by an authentication component. This component knows how to interact with a widely used Web service that can provide authentication information for its large user base. Each authenticated request is then processed to extract its specific query on the social network database. In order to decrease the response time of such requests, each query is first intercepted by a component that attempts to serve the query from a distributed cache of recent queries. When there is no recent result to such a query in the cache, an object-relational mapping component translates the programming language representation of the query into the query language of the underlying database management system and then converts the results back into corresponding programming language objects. These examples illustrate some of

the many steps involved in processing a Web service request and how middleware can facilitate their implementation.

Middleware has evolved to cater for an increasing number of concerns. Middleware was developed to support the implementation of distributed object-oriented software systems [Emmerich, 2000] and to support the development of distributed systems as service-oriented systems. The two dominant styles of service-oriented systems are SOAP-based and RESTful Web services. SOAP Web services are the result of industry efforts to support interoperability through standards and common middleware services. The common base language of these standards is the Extensible Markup Language (XML) [Bray et al., 2008]. The Web Service Description Language (WSDL) [Christensen et al., 2001] represents the interface definition language of SOAP Web services. Developers define the operations and data types of a Web service along with binding information that links the interface to a specific instance at a URL. Server and client stubs can be generated from the WSDL description of a Web service. These are responsible for binding to endpoints as well as generating and parsing the messages exchanged in invocations. The Simple Object Access Protocol (SOAP) [Mitra and Lafon, 2007] defines a standard message format for services to communicate with each other. SOAP messages consist of a header and a message body. The latter carries the message payload, which can be encoded according to application- or domain-specific XML Schema Definitions (XSD) [Walmsley and Fallside, 2004].

A number of specifications exist to define common middleware services for Web services, which we commonly refer to as WS-*. Examples of such middleware services are WS-Coordination (WS-C) [Feingold and Jeyaraman, 2009], which defines a decentralised coordination protocol on top of which standards such as WS-AtomicTransaction [Little and Wilkinson, 2009] have been implemented[1]. The composition of individual Web services into larger compositions is addressed by the Business Process Execution Language (BPEL) [Jordan and Evdemon, 2007]. BPEL is an XML-based workflow language that allows for the expression of control and data flow among Web services. It provides constructs for data manipulation, parallel invocation of services, loops and decision activities. Each BPEL process exposes its own Web service interface and can thus be composed hierarchically into larger orchestrations.

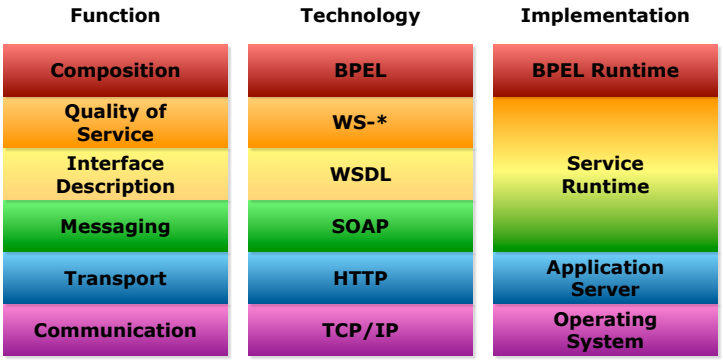


Figure 2.2: The SOAP Web services middleware stack and its typical implementation in Java EE and .NET runtime environments.

[1] We review these specifications in section 2.2

The components comprising a middleware-based application are conceptually divided into several *layers* that together form a so-called *software stack*. We categorise components as belonging to one of three layers. At the bottom of the stack is the operating system (OS) layer. This layer manages the hardware resources of a host, such as CPU, memory and network interfaces, and provides abstractions to execute processes using these resources. The next level up is the middleware layer. It consists of the kinds of components we have described in this section. In general, this layer is responsible for masking issues of heterogeneity and offers high-level abstractions, implemented using features of the underlying OS, to deal with various other concerns. The application layer comprises the actual application logic and builds on the middleware layer. We also refer to this layer as the service or composition level as it primarily consists of Web service interfaces and BPEL processes. Each layer can be further subdivided according to its core functionality.

The different functional layers that comprise these three abstract layers in the SOAP Web services stack are shown in Figure 2.2. Communication is typically handled via the TCP/IP implementation of the underlying OS. In modern enterprise middleware such as Java Platform Enterprise Edition (Java EE) [Oracle, 2011] and .NET [Microsoft, 2011] an application server often provides the runtime environment for server processes. The application server is responsible for managing resources (e.g., threads) on behalf of the applications deployed within it, enables service endpoints to interact through HTTP messages and can host other commonly used middleware components. The service runtime forms the core of service-oriented middleware and Web services as well as BPEL processes depend on it as it hosts their interfaces and maintains stubs that serve as entry points to their implementations. The service runtime processes incoming and outgoing messages, dispatches messages to the correct service instances and is the place where many of the decentralised middleware services are implemented, such as for the secure exchange of messages. A BPEL workflow engine forms the composition layer. It maintains the state of BPEL process instances as they are executed and relies on the service runtime for exchanging messages with partner services.

RESTful Web services represent an alternative architectural style based on the concept of Representational State Transfer (REST) [Fielding and Taylor, 2002]. RESTful Web services exploit existing and widely available World Wide Web (WWW) technologies and avoid some of the complexity of SOAP Web services. A RESTful Web service exposes a set of resources, which represent data of the underlying application, such as for example a list of conference participants and detailed information about each participant. Resources are uniquely addressable within a domain by uniform resource identifiers such as URLs. These resources can be manipulated by clients based on the vocabulary defined by the HTTP methods. For example, an HTTP GET request on a resource retrieves it and a PUT request creates a new instance. A server responds to requests with standard HTTP response codes. This is in contrast to SOAP Web services where each service defines its own vocabulary in terms of the operations of its interface. RESTful services do not require compilation of service interfaces into stubs and the basic necessary infrastructure is a simple Web server.

However, even though RESTful Web services are not typically implemented using extensive enter-

prise middleware frameworks, for non-trivial applications similar middlware services are used in their implementation. For example, middleware to manage load among a set of request processors and to improve the efficiency of data retrieval are used; as are components to facilitate the processing of request payloads in the form of XML or JavaScript Object Notation (JSON) [Crockford, 2006]. Some providers of RESTful services release client libraries in various programming languages. These libraries then offer methods that encapsulate entire sequences of HTTP requests on resources within higher-level methods, which is similar to the function fulfilled by stubs in SOAP Web services. RESTful services remain distributed systems that delegate many common tasks to middleware.

The benefits that middleware provides for the development of distributed software systems come at a price. It introduces complexity and has an impact on performance as each additional layer of components involved in processing a request incurs a certain performance overhead. Middleware-less systems (e.g., Domain Name System) are generally not subject to these disadvantages. In spite of this, developers consider the trade-offs involved in the use of middleware to be justified. Modern middleware facilitates the construction of distributed systems by offering higher-level abstractions that encapsulate considerable expertise in particular areas of distribution. These abstractions provide functionality to manage many aspects of distribution. This saves the time of application developers and enables them to focus more on application logic than on low-level concerns. As an increasing number of functions and areas of responsibility have been delegated to middleware, it has become what enables a broad range of programmers to develop complex applications that integrate widely distributed and heterogeneous resources. Therefore, it is important to investigate in how far it is possible to mitigate some of the drawbacks of middleware-based systems, such as their propensity for critical failures.

2.1.2 System-Level Failures

Service compositions can be affected by a wide variety of different failures. We focus on a particular type of failure, which we call system-level failures. It is important to define and characterise these types of failures in order to understand what kinds of conditions we want to detect automatically and assist operators in diagnosing.

A service composition is a software-intensive system, which is defined in [ISO/IEC/(IEEE), 2011] as a "*system in which software development and/or integration are dominant considerations*". In order to make this more specific, our working definition of *system* is loosely based on the one provided by [Avizienis et al., 2004, pp. 12, 13], which considers computer-based systems that consist of a number of different components. Some of these components need to interact with each other in order to fulfil some function. A component can be further decomposed into smaller subsystems that work together to implement the larger system.

For our purposes a *component* is any hardware or software module that encapsulates a major unit of functionality as it is typically needed for the implementation of a service composition. We consider components at a coarse-grained level of granularity. That is, we do not consider the procedures or methods in a program or even individual libraries as components. Instead, we consider components to be services at the service level, software modules at the middleware level and the resources and hardware

components managed by the operating system. We also consider major subsystems to be components. For example, components at the service-level are service interfaces and service composition artifacts, such as BPEL processes. At the middleware level we consider a database management system as well as the databases it manages as components. Similarly, an application server and its memory subsystem are related but separate components. And finally, at the level of the operating system we have components such as network interfaces, file systems, memory and CPU.

In this thesis, we assign one of two meanings to the term "system" depending on the context. We sometimes use it to refer to the *system layer* or *system level*, which is any set of components situated beneath the application or service level. The components at the system layer comprise the implementation and infrastructure of the service-level. This interpretation of system is in use in this section where we talk about the location of the origin of certain failures. However, when we talk about a service composition in general or about its structure and dependencies, we use system to refer to a service composition and its underlying infrastructure in its entirety.

A widely used definition of failures in computer systems is provided by [Avizienis et al., 2004]. Here, an *error* is defined as a deviation of some part of system state from its correct state and its cause is called a *fault*. An error can eventually lead to a *failure*, which is defined as the deviation of delivered service from correct service. For example, consider a bug that prevents a component from releasing all of the memory assigned during the execution of a particular function. The repeated invocation of this function leads to a decreased amount of free memory. Over time, all of the memory available to the component is exhausted and it is thus unable to respond to further requests. In this example, the software defect that prevents the component from releasing all memory is the fault leading to an error commonly known as a memory leak. The memory leak eventually causes a failure of the component.

Our definition of a *system-level failure* is a subtype of the above definition and differs from it in three ways. First, its effect is to prevent an application or a service from making progress and completing successfully unless the failure has been resolved. This is in contrast to causing some arbitrary deviation from specified service, such as for example a certain percentage of requests violating a response time objective. Second, a system-level failure is caused by error conditions in the middleware components, the operating system or the network. And finally, system-level failures are due to operational faults that occur at runtime.

Our definition of system-level failures omits the following types of failures. Input-dependent failures that are activated by certain points in the input space of an application need to be handled by domain experts who posses an intimate understanding of the application domain. An example of such a failure are parameters for which some computation will neither terminate successfully nor approach a useful approximation of a result. Design and application logic errors are outside the scope of this thesis and are generally addressed by the large body of literature on software testing.

There are several causes for system-level failures, one of which is the complexity of the underlying system. A service composition is supported by a system of systems. A large number of components may be involved in processing each request. These components encapsulate substantive expertise about the

performance of some function whose implementation is often not trivial. Each component is influenced by numerous configuration options to, for example, trade off its resource usage with its ability to handle concurrent load. In addition, there are dependencies among many of these components. For example, the credit card processor in our example composition uses a BPEL engine that relies on a service runtime to send, receive and process messages. The service runtime in turn depends on an application server to process XML data and to send and receive HTTP requests. The application server uses a programming language runtime, such as a Java Virtual Machine (JVM), which, in turn, makes use of the operating system implementation of sockets to stream data between endpoints. The application server is furthermore responsible to manage resources, such as threads and memory on behalf of the service runtime and workflow engine. Their resource usage is related to each others' workload. An operation of the workflow engine causes a sequence of actions at the service runtime and both sets of activities need to be satisfied by the application server acting as the resource manager. These factors lead to an increase in the complexity of software systems, which in turn creates ample opportunities for failures to arise within components as well as out of the interaction between them.

Another cause of system-level failures are sustained high workloads on some or all parts of the system. Over time such high demand leads to overload and in some cases eventually to resource exhaustion. Resource exhaustion usually follows as a consequence of sustained overload, which pushes resource usage to its limits and activates latent errors, such as for example otherwise harmless memory leaks. Under conditions of overload, resources are still available and the application continues to progress. However, as resources are constrained, performance usually begins to degrade noticeably and the rate of progress of the application becomes insufficient. Resource exhaustion leads to absolute degradation without any further progress being made by the application and usually causes one or more components to crash. For example, when the operating system has to deny the provision of an additional thread to the JVM executing an application server, this server usually reacts by ceasing operation. Other examples of resource exhaustion are full connection buffers, memory exhaustion, running out of disk space or file descriptors. High sustained workload can result from either a large number of concurrent clients or be due to a computationally expensive application.

Other common causes stem from the cross-domain nature of service compositions, where resources fail, become unavailable or where varying network conditions make them appear so. The Propaganda service composition integrates most of its services over wide-area networks. Network connectivity can also be asymmetric such that a host may see another one as unreachable, but a third one may not encounter this problem. Unavailability of a service is only a symptom of an underlying problem. It could be due to something as benign as routine maintenance or due to a more serious problem with its underlying infrastructure. An outage of the local application can be caused by failures of resources in remote administrative domains.

The propagation of system-level failures has two primary dimensions. One is the propagation across the layers of software components supporting parts of a service composition. The effects of system-level failures are not generally isolated or tolerated within individual components and hence tend to propagate

across the layers of the middleware stack until they reach the service level. This is not surprising, if we consider the interdependence of the various components. The other dimension is the propagation from the service level across domain boundaries along the chains of dependencies inherent to hierarchically composed service compositions. Many components are involved in fulfilling a request and an application typically requires correct operation from all these components.

Consider the case of Propaganda creating a profile for a new subscriber. It submits a request to the analytics service, which in turn delegates it to its deployment at the platform provider. There, the request is in turn processed by various components as we have explained previously and eventually reaches a component responsible for accessing a database. Unfortunately, the storage area network that hosts the file system used by the database instances of the analytics service has become partitioned from the network. The database access component tries to reach any of the replicas, but the underlying socket implementation cannot establish a connection to any of them and repeatedly informs its caller of this. Eventually, after several retries that database access component gives up and reports the failure to its caller at the application server. This then propagates back along the request processing chain until the client at the analytics service is informed of the failure (in case it has not timed out in the meantime), which in turn returns to the original invoker at Propaganda with an HTTP error code. The Propaganda service is prevented from creating a profile for its new user due to a network issue in an administrative domain it is not even aware of.

System-level failures often have drastic negative impact on a service composition and their consequences fall into one of two categories. They either result in abnormal termination of the application or reduce the service quality below an acceptable threshold. Abnormal termination means that the execution of a service composition is interrupted before a request has been successfully served. This is particularly expensive for long-running applications. Many failures result in severe performance degradation of the overall application, but given that service compositions are generally automated systems instead of interactive ones, only someone or something actively monitoring the system may notice that something is wrong and initiate an appropriate course of action. Slowdown is not necessarily due to an easily identifiable performance bottleneck, but may be due to failure conditions that affect system operation. For example, an application that mistakenly handles certain exceptional conditions as if they were of a transient nature, may keep repeating an exception-handling loop and continue retrying an invocation in vain and thus cause a considerable backlog of processing steps and requests. This may continue until an operator determines the cause of the problem and fixes it. System-level failures typically prevent an application from making progress.

The conditions leading to failures can be transient or non-transient. Even a transient condition can lead to abnormal termination of the application, if it has not been prepared to handle the unanticipated condition adequately. In general, any type of resource exhaustion is a non-transient condition as components do not tend to restore resources efficiently once load drops again. For example, if the social network API service has received such a high number of requests so as to saturate its web servers' request buffers, the web servers may recover once demand has dropped again for a while and there has

been a chance to process the backlog. However, if this peak demand caused some of the application servers that process the incoming requests to exhaust their threads or memory, they need to be restarted in order to recover these resources.

Repair actions to fix a system-level failure and reduce the likelihood of its recurrence usually fall within one of four categories. One remedial action consists of the fine-tuning of the many configuration options exposed by the various middleware components and the operating system. This may involve adjustment of the maximum thread pool sizes or an increase to the number of file descriptors the operating system grants to a single process. Other cases may require a structural change to the application. An example for such a repair action is to move the creation of a particularly large variable outside a frequently executed portion of the application. A third option is to distribute the various processes and supporting components differently in order to enable the system to sustain higher loads or reduce the frequency at which high numbers of large messages are exchanged between parts of the application. In some cases, rebooting is necessary to restore resources.

System-level failures cannot generally be dealt with effectively at the service- or application-level. They occur at a variety of places within a system onto which an application has no means to intervene in a directed and purposeful manner, even if all opportunities for system-level failures could be identified and anticipated at development time. After testing and fault tolerance techniques leave residual faults that materialise as system-level failures[2], the only remaining way in which to address these failures is often to detect their occurrence and identify their causes at runtime.

2.1.3 Detection and Diagnosis

In many cases, restoration of correct service can only proceed once detection of a failure and diagnosis to determine its causes have been completed successfully. These tasks are usually performed by a system operator who is responsible for keeping an application operational. Below we give two examples that provide some insight into how a system operator proceeds to detect and diagnose system-level failures. These examples illustrate part of the entire problem space an operator may have to investigate, before narrowing in on the actual cause of the problem.

Detection is the process of determining the presence of undesirable conditions that prevent an application or some of its underlying components from providing correct service. It constitutes the recognition that the effects of a failure are having negative impact on the operation of an application. Detection can either occur by end users who notice that they are not receiving service as expected, but preferably a system operator can detect failures before they have wide-reaching effects. In the absence of automated solutions to failure detection, a manual process demands that a system operator have some knowledge about what aspects of the behaviour of a component can indicate faulty behaviour, such as for example the rate of progress a component is able to make or its resource usage, and how to observe these aspects through relevant metrics. This is made difficult through the large number of different components that are typically present in a middleware-based software system and that a service composition relies on.

The system operator of Propaganda, Alice, has received complaints from several users about de-

[2]We review examples of such techniques in more detail in Section 2.2

creased responsiveness of the user interface. At first, Alice suspects an issue with the network connectivity to Propaganda's web servers at the compute service provider. She checks the request latency from various other servers situated at different geographic locations by logging into them and executing a few Unix shell commands. This is done to increase the likelihood of detecting asymmetric network issues that some, but not all clients may experience. Network connectivity appears to be normal and the web servers are available as Alice determines by sending them simple HTTP requests. Alice logs into the virtual machines deployed at the compute service provider in order to inspect these servers in more detail. The web server log files indicate an increasing number of requests, but the connection buffers still have sufficient capacity to store additional requests. Next, Alice starts the management console of the application servers handling the requests. She has to repeat this process for all the hosts that run application servers. She examines their resource usage and finds that even though the heap space usage is higher than what she expected it to be from experience, there is still some free heap space left. Accordingly, an exhaustion of the heap space by the application servers does not seem to explain the poor performance experienced by end users.

Alice has run out of ideas to check and her experience with this deployment does not point to any similar problems in the past that would not have become visible by the checks she has already performed. Therefore, she proceeds to examine several other system metrics in a random fashion in the hope that she will come across some suspicious behaviour. After a while, she notices very high CPU usage that appears not to be temporary on the hosts running the application servers. Through the use of some shell commands, Alice determines this to be due to extensive I/O activity. She finds that the operating system on these hosts has been making extensive use of the swap space. After examining the amount of free physical memory, Alice discovers a hypothesis that could explain the performance problem. The maximum heap space size of the application servers has been configured to be the size of the physical memory on these hosts. During times when the number of requests increases, the application severs require more heap space until they use up most of the available physical memory. As the application servers do not release their once claimed heap space allocation again, the operating system is forced to rely increasingly on its virtual memory. The disk I/O operations this involves severely degrade overall performance of the hosts. In order to test this hypothesis, Alice modifies the maximum amount of heap space, observes the system to determine whether there is indeed a positive impact on CPU usage and I/O activity on the affected hosts and then communicates with the previously affected users to enquire whether they perceive the problem as resolved.

On another occasion, on her periodic scans of various log files Alice detects that several recent payment transactions have failed. After examining the local infrastructure, she determines that the cause of the problem is not local and decides to email Bob, the system operator of the credit card processor, about the problem. Bob examines the log files of BPEL processes at the times of the failed transactions and finds that some of the processes had to retry their invocation to a particular bank's Web service, which was slow to respond at that time. Bob uses a test client to invoke the remote partner service several times and finds that its response time is considerably higher than usual. Whenever the initial

invocation times out, the BPEL processes invoke another local BPEL process that encapsulates some fault-handling behaviour to manage the continued interaction with the remote partner services. These repeated invocations each consume a single thread. The underlying runtime consumes several additional threads to wait for a callback on these invocations. Bob determines that as the backlog of processes waiting for the bank's Web service increases beyond a certain number, the combined number of threads consumed by the BPEL engine and the service runtime exceeds the number of threads available in the application server thread pool. The occurrence of this issue can be prevented by reducing the degree of concurrency with which both the BPEL engine and the service runtime can operate.

Service compositions do not posses facilities that would suitably aid with the detection of system-level failures. Timely detection of a failure requires continuous monitoring of system operation by a human operator. This involves the inspection of log files as well as other potential problem indicators, such as the availability of resources, the response times of endpoints and various resource usage metrics. There a number of log files from different components that each contain a large number of messages commensurate with the high number of recorded events. This has the effect of making it more difficult to spot messages which are potentially useful to detect a failure. For some failures, no extraordinary event may be recorded by any of the log files, even though the application is prevented from making further progress in a timely manner. Such a manual approach necessarily involves long periods during which no noteworthy observations are made, which is likely to negatively impact operator efficiency. Consequently, receipt of complaints by users of an application may constitute the first notification about the presence of a failure. Such complaints arrive after a failure had an opportunity to widely propagate its effects. Detection of failures in this manner is not only time-consuming and error-prone, but may also be too late to prevent wide-ranging outages.

The diagnosis of system-level failures is difficult as system operators are faced with a large problem space and few facilities to navigate it efficiently. *Diagnosis* can be regarded as consisting of the formation of hypotheses about the cause of a failure and then testing these hypotheses informally. The initial set of hypotheses, which may be an empty set that necessitates an inspection of random elements to gain further insight into what might have gone wrong, is usually based on an initial user complaint or observation made by the operator. The goal is then to reduce the set of wrong or inconsequential hypotheses and find one that captures the conditions that are in fact responsible for the failure. The problem space that an operator explores in this manner consists of the states of the large number of components involved in servicing a request. These states can be observed through various metrics about the operation of a component (e.g., log files, resource usage, performance metrics, etc.) and are influenced by concurrent workloads.

An operator collects and inspects data in order to test her hypotheses. This data is spread across various components and hosts and requires a variety of access mechanisms. For example, the operator may need to access information contained in the log file of a specific BPEL engine on one host, then measure the resource usage of its corresponding application server and log into another host to obtain statistics about a network interface. In the process the operator may make use of a variety of shell

commands to access data about various processes, issue HTTP requests to determine the availability of web servers and inspect log files. To decide which components to select for further investigation requires knowledge about the dependencies between an application and its underlying components as well as among these components. Knowledge about these dependencies is buried within a variety of sources, such as BPEL process definitions, deployment descriptors and configuration files.

In addition, a system operator needs to filter useful from inconsequential data. Some of the error reporting results in messages that do not necessarily contain information that helps with the diagnosis of a failure. For example, the error reporting is often based on HTTP error codes, which are wrapped into exceptions as a failure propagates upwards through the stack. Such messages usually indicate the unavailability of an endpoint, which merely points to the symptom of an underlying issue. Another issue is the lack of historic records of data about system operation. By default, only very few metrics are recorded permanently in order to avoid a great deal of unnecessary data from accumulating. However, this complicates the diagnosis process, as an operator may begin monitoring specific metrics only after the conditions leading to a failure have occurred. The lack of historic records makes it particularly challenging to diagnose the cause of transient failures.

Finally, the cross-domain nature of service compositions means that an operator may have limited visibility and control over some of the components that are critically important to the reliable operation of an application. It can be difficult to determine whether the cause of a failure lies within a remote administrative domain and successful diagnosis may require cooperation by the operator of the remote domain. This analysis suggests that we can improve on the current approach to the detection and diagnosis of system-level failures in service compositions.

2.2 Existing Approaches

Faults are addressed at various stages of the software lifecycle. Before actual coding begins, techniques for requirements elicitation and design reviews aim to prevent the introduction of faults. Using verification and validation techniques, software developers remove as many faults as possible before deployment. At runtime, numerous fault tolerance techniques enable software systems to overcome the effects of residual faults and continue processing. Software reliability engineering is concerned with developing quantitative models about the failure behaviour of software. To a large extent service compositions are comprised of ready-made software components. Requirements elicitation and design of these components are not within the control of a process modeller and measuring data about existing faults, as it is practiced in software reliability engineering [Lyu, 1996], does not directly help with keeping a large service composition operational. Therefore, in this section we have chosen to focus our review on approaches that help to reduce the occurrence of failures and enhance the ability of Web service compositions to tolerate them. In particular, our analysis determines to what extent existing testing and fault tolerance techniques can adequately address system-level failures.

2.2.1 Web Services Testing

The testing of applications that have been built using Web services presents several challenges. For one, there is the question of how one can support developer testing of Web services in the form of unit testing in an environment in which the tester may have only limited access over resources. Generally, we can only assume access to the interface of a Web service and not its implementation. This also raises the question how to generate useful test cases automatically given just this information. Third, the specification of a Web service is usually devoid of any behavioural specification, from which one might, for example, derive what constitutes a valid sequence of operation invocations. This shortcoming has resulted in work on model-based testing to check whether a set of Web services conform to valid protocols in their interactions.

A large body of work to address these and other issues of testing service-oriented systems exists. We can only present a brief overview of a few relevant ideas and refer the interested reader to [Bozkurt et al., 2010] for a relatively recent overview of the plethora of approaches. We limit our review to approaches that deal with three key issues: unit testing, specification-based test case generation and model-based conformance testing.

Unit testing is a fundamental technique for the quality assurance of software systems. Unit tests are generally devised by developers to test functional units of code. For example, for a Java program each method in a class should represent a well-encapsulated unit of functionality. Individual tests can be organised into larger test suites and there is usually tool support to automate their execution.

One of the challenges in unit testing Web services is how to test its functionality without access to its source code. Fault injection at the message level offers itself as a possible way. Network-level fault injection usually consists of random perturbations to messages, such as the reordering of the bits in the payload. In [Looker et al., 2007] the authors present a tool for testing SOAP-based Web services (WS-FIT) that extends network-level fault injection by allowing for more targeted and meaningful perturbations. A tester can specify scripts to not only delay or drop a particular message based on its content, but also to apply a particular modification to a specific parameter in an RPC message. WS-FIT works by instrumenting a SOAP runtime and intercepting incoming and outgoing messages. Each intercepted message is processed by a fault injection engine, which processes its contents to determine whether any of the injections specified in the scripts deployed by a tester should be triggered. It then performs the required injection, before passing the message on to its intended recipient.

The work presented in [Fugini et al., 2008] recognises that a tester may have access to the service implementation in some cases and may perform fault injection manually by modifying the source code or data in any underlying databases. If such access is not provided, the proposed approach instruments a SOAP runtime by inserting a message handler to intercept any messages. A fault generator can then inject a variety of faults. In particular, the authors propose to inject faults that simulate overloaded services through delayed messages and data faults in the form of typos, format inconsistencies, missing data and semantic conflicts.

The work presented in [Mayer and Lübke, 2006] addresses the unit testing of BPEL processes. The

authors discuss various concerns that are common to unit testing and that need to be implemented in order to enable testing of BPEL processes. These concerns include the specification of test cases, their organisation into larger test suites and the execution of the tests through tool support. The specification of tests is done entirely in XML. Testers supply information about the process to be tested as well as a set of mock services that represent its partner services during testing. Tests are defined in terms of the interactions between the process and its partner processes and the data that the process is to be invoked on during these interactions. Responses are checked against Boolean XPath [Clark and DeRose, 1999] conditions that the testers also specify. A prototype called BPELUnit processes the test specification and then deploys the process under test along with any mock services, executes the test via a generated test client, records the results and then undeploys the entire setup again. As such, BPELUnit provides tool support for the entire life cycle of unit testing a BPEL process.

The idea of testing services at the time of their registration with a directory service, such as a Universal Description, Discovery and Integration [Clement et al., 2005] (UDDI) server, is described in [Tsai et al., 2003]. The authors propose to extend UDDI registries with testing as an integral feature. Each Web service provider should, in addition to the WSDL specification of a service, also provide a series of test scripts to check the behaviour of their Web service. Such an enhanced UDDI registry would only admit a Web service, once it had passed all its test scripts. Similarly, clients could then execute any of the test scripts associated with a Web service in order to decide whether or not to use a particular Web service.

Another issue that is addressed by several approaches is how to generate useful test cases given only access to the interface of a service. Approaches to automate the generation of test cases usually belong to one of two categories. Specification-based approaches rely on features of the specification in order to create service requests that are already initialised with suitable data. Model-based approaches are an alternative to the ones we review next.

In [Sneed and Huang, 2006] the authors present a tool called WSDLTest, which addresses both test case generation and execution of tests. WSDLTest parses the WSDL document of the service under test in order to determine the XML Schema data types of its operations' parameters. It then generates random but compliant data values that can be used in valid service requests. Testers can override the randomly generated service requests through test scripts, which also allow for the specification of post-conditions to check invocation responses against. A BPEL process is used to execute the testing process by performing invocations according to a specified test script and then checks and stores the results.

The approach proposed in [Xu et al., 2005] is somewhat more sophisticated in how it generates test cases. Similarly as described above, the WSDL document of the service under test is parsed to determine the XML Schema data types of its operation parameters. However, the authors suggest to generate data based on a set of perturbation operators that create variations on the valid XML structures. The goal of test case generation in this case is to generate service requests that contain invalid XML data. The authors define operators for the deletion and insertion of nodes from an XML structure and one to modify constraints imposed by a data type. The latter can be used to generate boundary values that lie outside

the valid range specified by a constraint (i.e. maximum number of characters in a string). Beginning with an original and valid XML structure, the proposed approach generates one XML message per possible deletion and insertion as well as one message per constraint. While this approach can result in a great deal of test data, there is no guarantee that all messages generated in this manner are valid test cases.

Another approach to test case generation that focusses entirely on boundary values as a strategy to generate useful test cases is presented in [Zhang and Zhang, 2005]. The primitive XML data types of all operation parameters of the service under test are analysed. The generation of boundary values is then based on the values of the XML constraining facets for these data types. For example, a string in XML has facets that specify constraints over its minimum and maximum length. Based on this the proposed approach would generate strings that are shorter or longer than specified by the constraints, the empty string and so on. This approach can generate a great number of test cases in an automated manner by applying simple strategies for different types of constraints. Test cases that depend on the semantics of a parameter (the example given is how to generate useful tests for the expiration date of a credit card) still need to be generated manually.

The WS-TAXI tool described in [Bartolini et al., 2009] produces compliant test cases that should pass without raising any faults. WS-TAXI uses the WSDL document of the service under test as its input and generates a skeleton test case that does not yet contain any input data or declares what the expected outputs are. Test cases are generated based on a set of simple heuristics applied to the XML Schema data types of the operation parameters of the service. For example, when an XML choice element is encountered, test cases can be generated to enumerate all possible child elements. Other rules generate test cases to conform to boundary constraints for a data type. Apart from the requirement to specify sensible heuristics for data types, WS-TAXI generates compliant test cases in a largely automated manner.

Model-based approaches are an alternative to ones that rely on the specification of a Web service. Models often represent the desired behaviour of a service or a composition so as to afford checking whether a service conforms to correct protocols in its interaction with other services.

In [Foster et al., 2003] the authors describe an approach to perform model-checking of BPEL processes in order to find out whether their implementation gives rise to undesirable behaviour (e.g., deadlocks or any other deviation from specification). The authors propose do so by a comparison of two models with each other. One model specifies the desired behaviour and the other one corresponds to the implementation of the BPEL process. First, a developer specifies the desired behaviour of the BPEL process in the form of Message Sequence Charts, which capture the message exchanges between endpoints in a manner similar to sequence diagrams in UML. A tool then translates this specification into a Finite State Process [Magee and Kramer, 1999, Appendix B] (FSP) representation. FSP is a process algebra that has been used extensively to model concurrent processes. A model expressed in FSP can be translated into a state machine to facilitate checking its behaviour. Next, the developer implements the BPEL process according to its specification in MSC and maps the process into FSP. The authors provide a mapping from BPEL constructs into FSP for that purpose. The two FSP models that have been created

in this manner can be compiled into state machines and a corresponding tool can determine whether the possible traces produced by these two state machines are equivalent or not. If they are not, this indicates that either the specification is incomplete or that the implementation allows for undesirable behaviour. This approach demonstrates how model-checking can provide an opportunity to find out about potential problems even before deployment of the actual process.

In [Bertolino et al., 2006] the authors propose to test whether a Web service interacts correctly with other Web services in a UDDI registry at the time of its registration. The underlying model for this test is in the form of UML Protocol State Machines (PSM) [Object Mangement Group (OMG), 2002], which allow for the expression of ordering constraints over a sequence of operation invocations and the association of pre- and post-conditions with such invocations. A service provider is expected to supply a suitable PSM specification for a Web service that is to be registered. The UDDI registry then generates a sequence of test invocations according to the PSM. When the new Web service requests access to other Web services via the UDDI registry, it generates a proxy for the requested service and then in turn uses the PSM of the target service to check whether the invocations made to it by the new Web service are valid and in the correct order. Once a new Web service passes these tests, it is added to the UDDI registry along with its PSM specification.

There are many potentially useful representations for models of the desired behaviour of a service. In [Yan et al., 2006] the authors define a mapping from BPEL constructs to Extended Control Flow Graphs (XCFG) and demonstrate how the XCFG can be used to enumerate all feasible execution paths and assign valid values to the variables in a process through constraint solving. A constraint solver can generate valid test cases by finding values that fulfil the execution conditions of the determined paths.

These approaches support an important aspect of the quality assurance of Web services. However, they are not able to address may of the causes of system-level failures. Specification-based approaches check for syntactic and semantic errors in the interaction with a Web service. Similarly, model-based approaches are useful to check that services conform to desired behaviour and correct protocols. We do not cover approaches to test distributed systems here. Even if we were to assume testing of only the local infrastructure, this sill confronts a tester with a large number of components and requires expertise with all these to test them effectively and fix any faults. Instead, we assume that the developers of each middleware component have done their utmost to identify and remove as many faults as possible. Enabling the testing of Web services is important, but system-level failures are due to unforeseen and sometimes unavoidable operational conditions that need to be handled at the time of the occurrence of the conditions that give rise to them.

2.2.2 Fault Tolerance Techniques

2.2.2.1 Exception Handling

Exception handling mechanisms [Goodenough, 1975] enable a program to deal with exceptional conditions that may arise during its execution. An exception is typically a condition that would prevent some part of a program from producing a valid result. A programmer can specify code that checks for certain conditions in the execution of a program and instructs it on how to react to these conditions. This is

achieved by using exception handling constructs. In object-oriented programming languages a keyword such as *throw* or *raise* is used to indicate the occurrence of an exception in the execution of a method. For example, a programmer may add code to a method to detect invalid input parameters and raise an exception object of a particular type. This exception object can contain further information about the encountered condition. The programming language facilities then determine a matching exception handler and control is transferred from the code raising the exception to the handler. This handler encapsulates the logic necessary to deal with an exception of the given type. A matching exception handler is typically associated with the method that has raised the exception, but it may be necessary to propagate the exception further up the call stack, before it has been handled appropriately. Once the handler has completed successfully, control returns to the previous thread of execution, which may either terminate the failed operation or be able to resume it.

In applications that are built as compositions of SOAP Web services the application- or service-level is usually represented by a set of BPEL processes that together from a larger workflow. BPEL provides process modellers with three primitives to handle exceptional events encountered by their workflows. These constructs are referred to a as fault, compensation and termination handlers. Fault handling in a BPEL process is based on the assumption of immediate termination of ongoing activities and undoing already completed ones. The mechanism to compensate the activities of a BPEL process is based on the saga advanced transactional model [Garcia-Molina and Salem, 1987], which we discuss in more detail in the next section.

Activities in a BPEL process can be grouped into logical units of work by embedding them within scopes. A scope is similar to a programming language block. A process modeller can attach fault, compensation and termination handlers to a scope in order to catch faults raised during the execution of the scope's activities, compensate its completed activities and perform any actions required to cleanly terminate the scope. Compensation handlers contain activities that undo the effects of their corresponding scope. A compensation handler can only be invoked from within another compensation handler or fault handler. In BPEL, visibility of scope names and compensation handlers is limited to the immediately enclosing scope. This constrains the invocation of a compensation handler. A compensation handler for a scope B can only be invoked from a compensation or fault handler attached to a scope that immediately encloses scope B. In addition, while a fault handler is available for invocation from commencement of a scope, a compensation handler is only installed and thus available for invocation for a scope that has completed successfully. Fault handlers can either catch specific named faults as they are defined in the composed Web service interfaces or raised as standard faults by the underlying BPEL engine. Alternatively, it can be made to catch any fault that occurs in a scope. Faults can contain further data pertaining to the condition that occurred. In practice, this information is an XML representation of a programming language exception raised by the underlying service runtime implementation or BPEL engine. Any scope for which a fault handler was executed, is regarded to have terminated abnormally. Termination handlers can perform additional steps after the default termination of all activities has completed. In order to ensure compensation in cases where explicit handlers are missing or where faults are not the result

of an operation invocation, but instead represent an exceptional condition of the runtime environment, the BPEL specification provides rules for default compensation handling. Default compensation undoes successfully completed scopes in reverse order of their execution. This represents a basic overview of the rules governing fault handling in BPEL service compositions.

The rules governing fault and compensation handling in BPEL are complex. These rules are examined more closely in [Khalaf et al., 2009]. Based on a number of examples the authors demonstrate that the mix of explicit and default compensation handling as well as other features make it difficult for a process modeller to reason about process behaviour, if faults need to be handled. In [Greenfield et al., 2003] the authors identify different types of state that generally affect workflow processes, such as the real-world state, its abstract representation in a computer system and the current state of a BPEL process. They demonstrate how these types of state motivate additional fault handling capabilities based on a number of scenarios. For example, in some cases it may be necessary to temporarily pause a workflow to let a human being decide how to handle a situation. It should also be possible to let a process continue normally in case an alternative sequence of activities can achieve a previously failed objective. Some failed activities could be optional and do not need to cause a process to terminate. Additional processes can be executed to handle temporary failure conditions and enable the main process to continue in the meantime. And finally, it should be possible to restore process state to a savepoint and then redo execution from that point. [Greenfield et al., 2003] refers to the first verision of BPEL, known as BPEL4WS 1.0 [Curbera et al., 2002], but the obsevations remain equally valid for subsequent versions of BPEL, such as BPEL4WS 1.1 [Andrews et al., 2003] and the latest WS-BPEL 2.0 [Jordan and Evdemon, 2007]. Another source of exception handling patterns that are common to workflows and not necessarily easy to implement given the fault handling constructs of BPEL is presented in [Lerner et al., 2010]

There are numerous proposals to improve the existing fault handling capabilities of Web service compositions. [Khalaf et al., 2009] remove default fault and compensation handlers from BPEL processes and instead rely only on ones explicitly declared by a process modeller. Furthermore, handlers can be called from any ancestor scope. These changes help to simplify the standard rules. Another issue is how to achieve coordinated exception handling among a set of composed Web services in what is essentially an open, wide-area system consisting of autonomous sites. One proposal to address this issue ports Coordinated Atomic (CA) actions [Xu et al., 1995] to Web services in the form of Web Service Composition Actions (WSCA) [Gorbenko et al., 2007]. CA actions were originally proposed for use in object-oriented concurrent systems to enable different threads to achieve, among other things, synchronous exception handling. When an exception is raised within a CA action, all threads that belong to this CA action invoke exception handlers in a coordinated manner in order to resolve the issue. Composed Web services are structured within hierarchies of WSCAs. When an exception is raised that cannot be handled internally to an individual Web services, the WSCA attempts to handle the exception cooperatively among participants. Other proposals address the impossibility of identifying all potential faults at design-time by following a model-based exception handling approach. For example, in [Friedrich et al., 2010] process modellers supply information about data dependencies, correctness of

input and output data objects of the entire process and of individual activities and specify which one of three repair actions (i.e. retry, compensate, substitute) are applicable to each of the process activities. When a fault is raised, a planning algorithm is used to determine a sequence of repair actions that can restore the affected data objects to a valid state so that the process can continue.

Exception handling is similarly important for Web service compositions as it is for other types of software systems. It provides a way to address application-level faults as they are declared in the interfaces of composed services. Examples are faults to indicate an invalid item code in a purchase order or to inform the caller of depleted stock. Exception handling mechanisms can also handle some transient system-level failures successfully. A temporary network connectivity issue or unavailability of an endpoint can be countered by retrying a failed activity several times. In addition to enabling a service composition to overcome certain exceptional conditions, it provides programmers with an opportunity to identify and document what can be expected to go wrong with the invocation of a particular operation. Exception handling constructs separate normal business logic from exception handling code and their respective control flows at runtime. If appropriate logging mechanisms are in place, a record of exceptions can be maintained for later inspection and analysis.

Exception handling is not equally powerful for any type of exceptional condition. For example, overload or resource exhaustion events, which cause a middleware component to crash, often do not present an application that depends on this component with an opportunity to react in any way. And even if it did, the application is not the right place to deal with system-level failures. It possesses no facilities to directly intervene in specific middleware components and neither should it have to know how to influence middleware in a certain way. The underlying layers are hidden from the application to a large extent in order to encapsulate implementation details and provide transparency. Furthermore, for some system-level failures no exceptions are raised at all, even though they result in severely degraded performance. Any advanced exception handling mechanisms one could envision that would be able to deal with system-level failures more intelligently and have a broader array of options for countering their effects will probably need to be based on facilities that provide more detailed and actionable information about a failure and its potential causes. Existing mechanisms based on exception handling are primarily useful for handling application-level faults, but less able to effectively deal with system-level failures as we consider them.

2.2.2.2 Transactional Mechanisms

Our discussion of transactional mechanisms is divided into traditional and advanced transactional models. Transactions, which form the basis of database management systems (DBMS) [Gray, 1978], provide a useful abstraction to maintain consistency in spite of failures. The traditional transactional model is based on the four properties of atomicity, consistency, isolation and durability (ACID) [Haerder and Reuter, 1983]. Ensuring atomicity demands that either all actions belonging to a transaction are executed or that the results of none remain in effect. A transactional system guarantees that the effects of each transaction leave the system in a valid and consistent state. Isolation means that the effects of concurrent, not yet committed transactions remain hidden from each other. Durability

refers to the persistent storage of the effects of a committed transaction so that they survive any failures. The implementation of the ACID properties is primarily based on the locking of shared resources, the maintenance of logs and the durable storage of committed changes. Locking of resources prevents any inconsistencies that might otherwise be introduced by concurrent transactions accessing shared resources. Transaction logs record data about each action of a transaction in order to enable backward and forward error recovery to be performed in an automated manner.

The success of the ACID model is due to the fact that it effectively handles concurrency and failures on behalf of a programmer. For example, a DBMS schedules transactions to increase concurrency while ensuring no incompatible data modifications can take place. Furthermore, it can react to failures by retrying a transaction or undoing its effects. A transaction processing system that implements the ACID properties effectively manages concurrency and failures on behalf of programmers to ensure consistency is maintained and enable programmers to focus on producing business logic.

Support for ACID transactions is as important for service compositions as it is for other types of software systems. However its applicability is limited by the characteristics of Web services. The interactions between Web services can be long-running and involve different administrative domains. In such settings a transaction coordinator that manages a transaction spanning a set of distributed participants would need to hold locks on shared resources for the duration of a transaction in order to maintain the ACID properties. However, locking of shared resources for long-running transactions in distributed settings is inefficient as these resources would become unavailable to other transactions for potentially long periods of time. Furthermore, in cross-organisational settings, organisations would need to allow each other or a third party to hold locks on their resources. ACID transactions are not suitable for long-running transactions that involve distributed sets of participants.

Similar issues were encountered by researchers long before the advent of Web services. ACID properties were found to be unsuitable for multidatabase systems (MDBS) and data-intensive interactive applications, such as computer-supported cooperative work applications (e.g., CAD/CAM, software development environments). MDBSs are distributed databases comprised of multiple autonomous sites, which need to coordinate the handling of transactions. Advanced database applications feature long-lived, user-controlled and sometimes interactive transactions. The autonomy of different sites together with the long-running and sometimes interactive nature of some transactions are not a good fit for the traditional transaction model. Maintaining strict atomicity via a global atomic commit protocol is inefficient given the communication delays and failure-proneness of a distributed setting. These challenges in the application of ACID transactions have led to the development of several advanced transactional mechanisms (ATM). ATMs make use of nested transactions in order to divide a long-lived global transaction into a number of subtransactions. They then make use of knowledge about the transactional characteristics of subtransactions in one of several ways so as to afford the relaxation of atomicity for these subtransactions. ATMs aim to reduce the amount of time resources need to remain locked and increase the degree of interleaving between subtransactions belonging to different global transactions. A more comprehensive survey of ATMs than we can afford here is given in [Elmagarmid, 1992]. Another

survey that focuses in particular on the shortcomings of traditional transactions for cooperative work applications and reviews suitable ATMs, can be found in [Barghouti and Kaiser, 1991]. We focus our review on some of the ATMs that have had influence on service compositions and workflows.

One early advanced transactional model exploits semantic information about the transactions performed by a particular application in order to relax atomicity [Garcia-Molina, 1983]. A long-lived transaction is divided into a number of subtransactions and programmers need to provide the DBMS with two types of information about these subtransactions. They assign semantic types to each subtransaction and indicate which types are compatible with each other and can thus be interleaved. Programmers also implement compensating transactions and assign one to each subtransaction. A compensating transaction is a series of countersteps that undoes any effects of an interrupted subtransaction. Based on this information the DBMS can achieve a higher degree of concurrency than would be possible with strictly serialisable schedules and through the compensating transactions is still able to provide a consistency guarantee referred to as semantic atomicity. That is, either all subtransactions of a global transaction commit successfully or the effects of any subtransactions up to the occurrence of a failure are undone based on a series of user-specified countersteps.

These concepts found implemenation in the form of sagas [Garcia-Molina and Salem, 1987]. A saga is a long-lived transaction that is executed as a set of nested transactions with associated compensation actions so that the DBMS can guarantee semantic atomicity. The classic example application of sagas is the booking of a block of seats on a passenger airplane. Several such block reservations can be executed in parallel and should one of them fail, the corresponding compensating transaction will perform a cancellation of the reservation for this particular group of seats. The other concurrent transactions do not need to be backed out in a cascading manner. Even though the state after compensation of the failed transaction is not necessarily the same as before this transaction began, due to other concurrent bookings and cancellations, it is consistent within the semantics of this particular application. sagas effectively relax atomicity and thereby increase concurrency, while still enabling a DBMS to handle failures and maintain consistency.

Numerous extensions and variations have been proposed to this early model. For example, [Elmagarmid et al., 1990] takes account of the fact that not all transactions may be compensatable and introduces the concept of mixed transactions that can be comprised of both compensatable and non-compensatable transactions. All compensatable subtransactions can commit even before their global transaction does and only non-compensatable subtransactions must wait. This idea was taken further to establish three subtransaction types in [Mehrotra et al., 1992]. A subtransaction can be either compensatable, retriable or a pivot. When a failure occurs, a retriable transaction can be retried and is guaranteed to succeed eventually. A pivot transaction is neither compensatable nor retriable. These types enable a transaction manager to manage concurrency by preventing a transaction from accessing data objects that another transaction may still compensate. Based on these transaction types [Zhang et al., 1994], among other contributions, provide a definition of well-formed transactions. In a simplified account, a well-formed global transaction consists of a single pivot, which is preceded only by compensatable and

followed only by retriable subtransactions. If a failure occurs before the pivot has committed, it can be aborted and all previously committed subtransactions can be compensated. Otherwise, once the pivot has committed, the global transaction is by definition guaranteed to eventually commit successfully. A DBMS can automatically verify whether a given transaction is well-formed and thus recognise transactions that may under certain circumstances violate consistency. These models have found application outside of database systems.

The lack of support in workflow management systems (WFMS) for the reliable execution of complex workflows has led some to propose solutions based on advanced transactional models [Georgakopoulos et al., 1995]. The motivation for the use of transactional mechanisms is to enable WFMSs to provide some runtime support for failure handling and relieve programmers from having to implement all of the failure handling logic themselves. For that purpose, [Hagen and Alonso, 2000] apply the concept of well-formedness to workflow processes and couple it with exception handling mechanisms. A well-formed workflow process consists of single pivot that is preceded by activities that can be compensated and followed by activities that can be retried any number of times. Exception handlers are associated with blocks of activities and take over control in case of a failure. An exception handler can then make use of compensating tasks to undo the effects of activities or resume the interrupted block of activities in order to retry a failed action. Transactional mechanisms have become an integral part of WFMSs and these ideas have also found application in Web services and in BPEL.

There are two primary specifications that define transactional support for interactions between services in the SOAP stack. Both are based on WS-Coordination (WS-C) [Feingold and Jeyaraman, 2009]. WS-C is an extensible framework that enables participants to agree on a particular protocol to follow by exchanging protocol-specific Web service interfaces. These interfaces define operations and messages that participants can use to follow the corresponding protocols. WS-C supports distributed coordination, in which participants communicate with each other via local coordinators that act as proxies. The transactional protocols based on WS-C are WS-AtomicTransaction (WS-AT) [Little and Wilkinson, 2009] and WS-BusinessActivity (WS-BA) [Freund and Little, 2009]. The former provides protocols to support services in the preservation of ACID transactions. For example, one of the supported protocols is two-phase commit. WS-BA caters for long-lived transactions that make use of compensating activities. The protocols and their interfaces merely inform participants of the progress and eventually of the outcome of a transaction. The implementation of corresponding behaviour is the responsibility of the Web service developer, who has to decide what it means to commit or abort in a particular application.

Transactions represent a useful mechanism to ensure consistency in light of concurrency and failures on behalf of programmers. The utility of transactions has resulted in their ubiquitous adoption in modern application servers. The saga ATM forms the basis for BPEL's implementation of compensation-handling, which we have already evaluated in Section 2.2.2.1. The application of ATMs to Web service compositions is not always straightforward. For example, in some cases it may not be clear how to assign transactional characteristics to heterogeneous Web service activities. There is also the difficulty of verifying that an overall composition maintains well-formedness without requiring service providers

to reveal sensitive implementation details about the component Web services [Ye et al., 2009]. More significant for our discussion is that neither ACID transactions nor ATMs can enable a system to tolerate system-level failures or provide any facility to prevent their occurrence in the future. They prevent the introduction of inconsistency by failures that interrupt critical activities, but cannot support human operators with the identification of the likely causes of a failure. As such, the kind of service transactions can provide is orthogonal to what we argue is required to handle system-level failures. Our goal is to improve the failure management capabilities of service compositions in a manner that enables system operators to return a system to normal operation through a more principled diagnosis process. This is not what transactions have been designed for.

2.2.2.3 Failure Detectors

In order to maintain the liveness and safety of a set of processes that communicate by message passing in spite of failures, it is necessary to be able to detect failed processes. Consider the example of a process p that sends a message m to another process q via a reliable transport protocol (e.g., TCP). Process p will continue to retransmit the message m until it receives an acknowledgement from q that confirms the receipt. Liveness at process p is maintained as it ceases to resend m, once it has received the acknowledgement. If we also allow for process crashes, an acknowledgement may not be forthcoming. Therefore, most reliable transport protocols will stop retransmissions after a certain number of apparently unsuccessful attempts. As it is not possible to distinguish between a crashed process and one that is merely very slow in an asynchronous system without timing assumptions, this can lead to a situation in which the safety of the sending process is violated. Failure to receive an acknowledgement may cause process p to pursue an alternative control path under the assumption that process q has crashed. However, q may actually have been alive and may have already acted upon the send message.

Failure detectors [Chandra and Toueg, 1996] enable processes to determine which other processes seem to have failed. A failure detector is characterised by its completeness and accuracy. The former describes its ability to detect failed processes and the latter is its false positive rate. A reliable failure detector is absolutely complete and accurate and as such could guarantee agreement among a set of processes on which ones have failed. It has been demonstrated [Fischer et al., 1985] that it is impossible to guarantee consensus in an asynchronous system with even a single process failure. This suggests that it is not possible to implement a reliable failure detector under realistic assumptions.

An unreliable failure detector [Chandra and Toueg, 1996] can overcome this issue under certain circumstances at the expense of accuracy. That is, an unreliable failure detector may incorrectly suspect that a process has failed and subsequently declare it to be alive again. An unreliable failure detector that is strong enough to solve consensus has the following properties. It guarantees that every crashed process will eventually be suspected permanently by every non-faulty process. Furthermore, some available process will eventually never be suspected.

There are several ways to implement an unreliable failure detector, but typically there is a failure detector module that is local to each process and continuously outputs a list of processes that it suspects to have failed. In one common approach this determination is based on the use of time outs. For example,

a process q sends a message that it is still alive to all other processes every t seconds. If another process p has not received such a message from q within time t plus some time that indicates some upper bound on the expected message transmission time, then the local failure detector at process p will add process q to its list of suspected processes. When the local failure detector at process p subsequently finds that it has suspected q incorrectly, it will increase the time out value so as to avoid the same mistake in the future. With sufficiently long time out values, the view of the system state stays stable for long enough to reach agreement.

For timeout-based failure detectors, there is a trade-off between detection time and detection accuracy. Longer time out values increase accuracy, but at the same time lead to higher detection times. On the other hand, shorter time out values can lead to higher false positive rates and can cause unnecessary disruption to a system. In [Stelling et al., 1998] the authors experiment with different time out values for unreliable failure detectors in a wide-area setting that consists of nine nodes. They find that a time out value of 240 seconds allowed their failure detector to achieve a false positive rate of $1/100,000$ and at 35 seconds the false positive rate was still only a fraction of 1%.

Another way to implement unreliable failure detectors that does not require the use of time outs has been proposed in [Aguilera. et al., 2000]. As is the case for time out-based failure detectors processes send heartbeats to all other processes. The local failure detector modules, however, do not list suspected processes, but instead they maintain a vector of counters; one per process. Each process that receives a heartbeat for another process increases its copy of the counter for this process. If the counter for a process ceases to increase, it is assumed to have failed. Consider the following example, in which process p sends a message m to another process q. As long as the counter for q at process p increases, p will continue to resend m. Once process p receives an acknowledgement or if the counter for q has stopped to increase, p can cease to retransmit.

Failure detectors provide an important service for the implementation of reliable distributed systems by enabling processes to detect and agree on which other processes have failed. However, this service is not sufficient to adequately detect all system-level failures for a number of reasons. First, it only detects that a process is unavailable and does not deal with less clear-cut failure conditions. The fact that a component is alive and sending heartbeats, does not imply that it operates as it should do. Second, in the multi-tiered architectures that is commonly in place in middleware-based systems, a large number of individual components per node would need to send and process heartbeats. Finally, highly accurate failure detection is possible even across wide-area networks, but only if an application can tolerate relatively long detection times[3].

2.2.2.4 Process Groups

Process groups represent a useful abstraction to support the implementation of consistent behaviour in safety-critical distributed systems. The Isis toolkit [Birman and Renesse, 1994] provides functionality to applications that facilitates their implementation through process groups. The two main components of

[3] An exception to this, at least for some failures, is presented in recent work on a data-driven failure detector [Leners et al., 2011].

process groups are group communication primitives and group membership services. A process joins a group, then interacts with the other members via some form of multicast and eventually leaves the group. Group communication primitives provide certain delivery guarantees for messages within a group, such that all messages are delivered in the same order by all processes and exactly once. A group membership service ensures that all processes have a consistent view of the current membership. In Isis, any process that has become unresponsive and is suspected to have failed is forced to leave the group so that the remaining members are prevented from interacting with the suspected process. A process that is suspected in this manner is then required to rejoin the group as a new process and is only integrated back into the group once it has updated itself with the current state of the group.

In [Vogels and Re, 2003] the authors propose to port process group semantics to Web services. WS-Membership (WS-M) is described as a potential extension to an existing Web service coordination framework (WS-Coordination [Feingold and Jeyaraman, 2009]). WS-M provides a membership service that is similar to what Isis offers. A so-called Service Tracker component maintains an up-to-date membership list for a group of Web services. Web services are expected to submit periodic heartbeats and the absence of a heartbeat for a Web service beyond a certain amount of time causes it to be regarded as failed and it is consequently removed from the group. Taking into account the cross-domain nature of Web services, heartbeats are exchanged using a gossip-style protocol, which affords an eventually consistent view of group membership, if there are no further updates to membership for a "long enough" period of time.

Process group semantics provide an important service for safety-critical distributed systems by providing a way to achieve consistent behaviour among a set of processes in light of process crashes and network partitions. However, the provided service addresses how to enable consistent coordination among a set of processes and as such is orthogonal to approaches that deal with the detection and diagnosis of the issues that underlie the failure of a process.

2.2.2.5 Redundancy and Design Diversity

Redundancy of hardware and software components is a widely used approach to improve the availability of software systems. For example, a set of servers can be arranged into an array of replicas. Upon the failure of one server, it is possible to automatically fall back on one of several backups. Alternatively, requests can be dispatched and processed by several replicas in parallel and the first one to complete, replies to the client. One weakness of such an approach is that coincident failures can quickly eliminate an entire array of replicas, if they all suffer from the same faults.

Design diversity is one way in which this issue can be addressed. The idea is to provide several functionally equivalent alternatives under the assumption that each will fail independently of the others. In [Randell, 1975] the author introduces the concept of recovery blocks as a way to structure programs for increased resiliency to failures. A programmer associates several alternate blocks of statements to a primary one. If the execution of the primary block fails, one of the alternatives is executed and an acceptance test is applied to determine, if its output is acceptable. Should it fail this test or fail in its execution, then the next alternative block can be executed.

N-Version Programming (NVP) [Chen and Avizienis, 1995] is an approach that relies directly on the potential benefits of design diversity. NVP proposes that several functionally equivalent variants should be developed from the same specification, but by different developers. The assumption is that the faults in the resulting variants will be independent from each other. At runtime, a coordinator synchronises the execution of the variants. It then matches their output to determine whether all variants reach the same result or, if they do not agree, it chooses the majority vote as the result.

In [Chan et al., 2006], the authors argue that the high cost associated with independently developing functionally equivalent components will be reduced with service-oriented computing, because different service providers will offer similar services. In anticipation of this scenario, the authors experimentally evaluate the effectiveness of several replication strategies for their ability to reduce the failure rate of a Web service. The strategies they examine are a single server without any replication, temporal replication in the form of client retries and server reboots, spatial redundancy, in which a request is dispatched to one of several replicas in case the primary fails and finally a hybrid approach that combines spatial and temporal redundancy. The most significant reduction in failure rates, at 9%, is observed for the hybrid approach.

Redundancy is an attractive approach to attempt to increase availability due to its simplicity. As computational resources become increasingly commoditised, the cost of extensive redundancy may also become less of a factor. However, redundancy is defeated by coincident failures. The cost of implementing design diversity is more considerable and it is not clear whether the vision of a large number of functionally equivalent services that can be used interchangeably by clients will be realised. Finally, in [Knight and Leveson, 1986] the authors experimentally evaluate the assumption that faults in separately developed programs are indeed independent from each other, but were not able to confirm this assumption. Redundancy and design diversity represent another important piece in the tool set for improving the availability of software systems, but is not sufficient to adequately address system-level failures.

2.2.2.6 Message Queuing

An important requirement in middleware-based distributed systems is the reliable exchange of messages between remote endpoints. Message queuing is a common technique to this effect. An application passes a message for a remote endpoint to a message queue, which then delivers the message to the queue at the remote endpoint with certain guarantees.

WS-Reliable Messaging (WS-RM) [Davis et al., 2009] specifies how such a message queuing service should work for the reliable exchange of SOAP messages between Web services. Endpoints can agree on several delivery guarantees, such as ordered message delivery, at-least-once, at-most-once and exactly-once delivery. The specification of WS-RM does neither prescribe a transport protocol to be used nor how the attainment of these delivery characteristics should be implemented. Message queues that persist messages, assign suitable identifiers and then continuously attempt to deliver them to the queue at the remote endpoint suggest themselves for this purpose.

Message queuing is an important and widely-used service in middleware-based distributed systems

as it simplifies an important reliability concern on behalf of developers. However, the reliability guarantee that it provides is that messages will be delivered to their endpoints eventually. As we have argued in this chapter, while the transparency such an approach provides facilitates the development of distributed systems, it is often desirable to detect failures as early as possible after their activation and to provide support to diagnose their causes in an efficient manner. The service provided by reliable messaging is orthogonal to these concerns.

2.2.2.7 Autonomic Computing Approaches

Autonomic Computing [Kephart and Chess, 2003] has been proposed as a way to improve the manageability of information systems by enabling them to become increasingly self-managing. The general idea is that systems should monitor their operation and environment, analyse this data and then perform various corrections as required to maintain a desired level of quality of service.

In [Bausch, 2004], the author presents a tool set to support non-expert users with the management of complex Grid computing applications that are expressed as workflows through autonomic mechanisms. One of the presented contributions is concerned with autonomic failure handling. The state of executing processes as it is captured by a process engine is persisted at regular intervals. If a process gets interrupted for some reason, it can be restarted from its last checkpoint. An algorithm ensures that recovery from persistent storage proceeds correctly. This algorithm identifies any orphaned compute jobs and resubmits jobs as needed to enable the correct continuation of execution. The proposed framework also allows process modellers to specify alternative services, which the engine can try to invoke in case the original service has failed.

In [Baresi and Guinea, 2011], the authors describe a framework for self-supervising BPEL processes. Their approach is based on the expression of monitoring and recovery directives in an XML-based language. Developers are expected to define pre- and post-conditions on the activities of a BPEL process. The monitoring directives are similar to assertions in a programming language and can be used to check various properties, such as for example whether the format of a returned identifier is as expected. A supervision manager monitors the execution of processes and intercepts any messages. It is thus able to evaluate the monitoring directives and, if an assertion should fail, trigger the execution of recovery actions that have been associated with this assertion. Basic recovery actions include rebinding to an alternative service, calling a set of services along with instructions on how to transform their output, retrying a failed invocation a number of times and notifying a user. Such basic actions can be composed into more complex recovery strategies. In this way, an assertion might for example examine the resolution of returned image data. In case the supervision manager determines that the resolution is too high, it could submit it to a service that can rescale this data. If the resolution was too low, an alternative service might be invoked on the original data to obtain a higher-resolution image. Finally, if all else fails, the supervision manager can notify a system operator about a problem.

System operators sometimes attempt to resolve failures by rebooting components. Rebooting restores an application and the resources it uses to their initial state and can, for some failures, restore service for some time without necessitating an immediate diagnosis and detailed fix. However, to reboot

an entire server is time-consuming and can incur considerable data loss. In [Candea et al., 2004], the authors propose so-called microreboots as a more fine-grained technique that reboots the smallest possible components first. The authors extend a Java application server to enable it to microreboot individual EJBs. They use simple failure detection mechanisms, such as parsing returned HTML pages for information indicating a failure, in order to trigger microreboots. A recovery manager listens to these failure reports and then tries to reboot the component with the cheapest recovery first. Only if this does not solve the failure, does it attempt to reboot progressively larger sets of components. In order for microreboots to be suitable, an application needs to have certain characteristics. Applications need to consist of small and decoupled components, important state needs to be separated from application logic and there needs to be a mechanism that can transparently retry requests while a component is being restarted. The authors are able to demonstrate that under these conditions, microreboots can be an order of magnitude faster than standard reboots that target an entire server.

Software rejuvenation [Huang et al., 1995] is similar in its motivation. By periodically restarting an application in a controlled manner, it should be possible to remove the effects of any resource leaks. If the restarts are performed at appropriate times, then the use of such planned downtime should lead to a decrease in overall downtime and a corresponding increase in availability. Rejuvenation intervals can be chosen based on experience with an application about when it approaches resource exhaustion. Alternatively, in [Vaidyanathan and Trivedi, 1999] the authors propose a measurement-based approach. This approach monitors several resource usage metrics in order to take the current workload into account for deciding when restarts should be triggered. This can avoid unnecessary disruption due to premature rejuvenation. While microreboots are a reactive strategy that is triggered upon detection of a failure, software rejuvenation is its preventative counterpart in that it is applied before a failure that is activated by resource leaks has an opportunity to take effect.

Generic approaches such as the ones just described can work effectively for some failures and are elegant due to their ability to be automated. However, frequent outages of large service providers probably bear witness to the fact that not all failures can be adequately addressed in this way. Some system-level failures require repair actions such as configuration changes or modifications to the structure and deployment of an application in order to be resolved permanently. If not all failures can be addressed in a generic manner, then determining how to support system operators with detection and diagnosis may be a first step to find necessary and sufficient metrics that can inform future autonomic mechanisms with more actionable information about failures. In any case, these approaches represents useful tools, but they do cannot remove the need for failure detection and diagnosis.

2.3 Our Approach: Data-Driven Detection and Diagnosis

Middleware has enabled a broad range of developers to build applications in the form of large service compositions. Middleware offers higher-level abstractions to developers and thereby creates the transparency necessary to hide much of the underlying complexity. However, underneath the service level we are dealing with complex distributed software systems that provide ample opportunities for failures as the previous sections in this chapter have illustrated.

Fault removal and fault tolerance techniques are important tools to improve the quality of software systems. However, several characteristics of service compositions limit the effectiveness of these techniques. We cannot assume access to the source code of most components or of the service implementations. And even if the source code was available, we are still faced with testing and fortifying a system of systems that consists of large numbers of interacting components. Most of these components have been developed by different parties and encapsulate non-trivial functionality and substantial expertise. It would be prohibitively expensive to test all interactions among a large number of components under a broad range of different workloads and ensure that all conditions that could lead to system-level failures at runtime had been covered. The developers of these components already expose their creations to extensive testing and still residual defects remain and can become active under some of the multitude of usage profiles. In addition, from the viewpoint of a service composition there is a lack of visibility and control over some components that are critically important to its reliable operation, but that are hidden behind published service interfaces. The space of possible causes for system-level failures is quite large. It appears that residual faults will always remain and can become active at runtime.

Fault tolerance techniques provide a type of service that is vital for many mission-critical applications. But for certain applications this service may not be adequate. For example, some approaches ensure consistency in light of failures by reacting with backward error recovery, which can be inappropriate in cases where a great deal of computation would be lost. Other techniques guarantee that a desired outcome will be reached eventually (e.g., the guarantee of eventual delivery of a message provided by message-oriented middleware). And yet other approaches enable a set of processes to reach agreement on which processes appear to have failed in order to avoid inconsistent behaviour among the remaining processes. Such approaches do simply not cater for the quick detection of a system-level failure and the determination of its causes. In some instances, fault tolerance techniques may fail to address system-level failures. For example, certain failures can quickly eliminate several replicas, all of which are subject to the same vulnerability. In the end, system-level failures often tend to persist and consequently service compositions are faced with severe failures that need to be handled at runtime.

In addition to the limitations identified in our review, the shortcomings of existing approaches for dealing with system-level failures is evidenced by prominent outages of large Internet services, some of which we mentioned in the previous chapter, where even after rigorous testing and the application of state-of-the-art fault tolerance techniques system-level failures do occur and cause wide-ranging effects.

It is important to develop a more principled manner in which to manage the occurrence of such unavoidable and severe failures at runtime. The sooner we can detect a failure and determine its causes, the faster we can fix it and restore correct service.

For this purpose we propose a data-driven approach to the detection and diagnosis of system-level failures in middleware-based, cross-domain distributed software systems as they form the basis for service compositions. An overview of our approach is shown in Figure 2.3. It stands in contrast to the common one, which is based more strongly on the experience and even intuition of a system operator. The aim of our approach is to improve the process of detection and diagnosis of system-level failures by

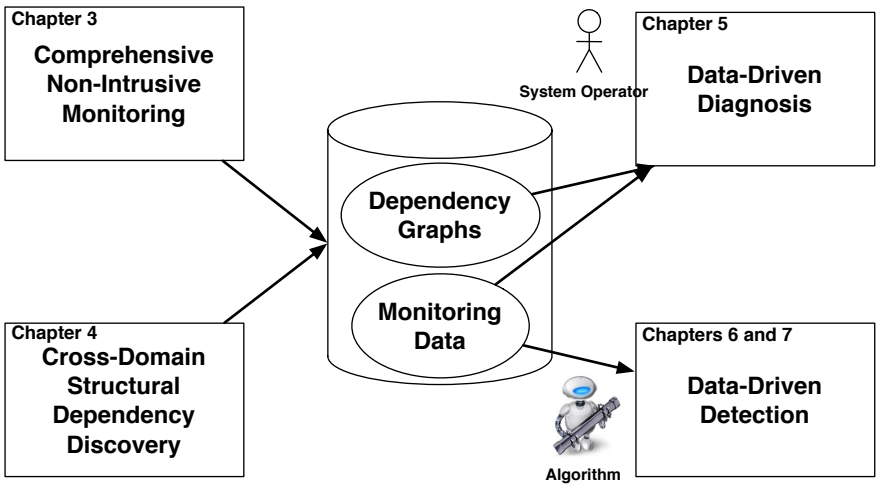


Figure 2.3: Overview of data-driven detection and diagnosis.

basing it on data about system operation and structure.

While existing monitoring systems, some of which we review in Section 3.6, are typically available to system operators, they may either focus on high-level metrics that are useful to evaluate the performance of a business process or they may be configured to monitor a small number of key metrics in order to minimise their overhead. Additional monitoring may only be installed in response to problems with the operation of a system. We want to find out what the effects on the detection and diagnosis of failures can be, if we monitor a broad array of metrics about system operation and derive information about system structure as it is possible to do in middleware-based service compositions. We want to understand when such an approach is justified and beneficial.

We obtain such data through monitoring and dependency discovery. Monitoring involves the measurement of various metrics about system operation and in particular of metrics that can be observed without requiring intrusive instrumentation. This is important given that most relevant software components have not been designed to facilitate their instrumentation. As relevant data about system operation is spread out all over a system, it is necessary to integrate it. The prototype of our approach integrates data obtained from the service-level, the middleware and the operating system and it does so across hosts and, to some extent, administrative domain boundaries. Monitoring provides us with a repository of historic records of measurements about the operation of all components that make up a service composition within each administrative domain. Such records promise to facilitate the recognition of trends and anomalies and provide a unified view of a system's operation. Furthermore, it affords the correlation of measurements at the lower layers with application activity.

In addition to monitoring data, our approach exploits available sources of deployment information in order to discover dependencies among components from the viewpoint of a service composition. Dependency discovery provides us with dependency graphs that show the relationships among the components that comprise a service composition. This makes system structure more explicit and can serve to guide diagnosis along relevant paths of dependencies. The data made available from monitoring and the discovery of dependencies enables us to determine experimentally what the costs, effects and benefits of

such a data-driven approach are in practice.

The hypotheses we want to test about this data-driven approach are two-fold. First, we expect that this data will support system operators to engage in a more structured diagnosis process and consequently diagnose more failures correctly in a shorter amount of time. Second, as such data contains what effectively enables human system operators to determine the causes of failures, it should in principle be possible to achieve fast and accurate failure detection in an automated manner based on an analysis of this data.

Conclusions

This chapter has presented the motivation behind our work. We have outlined the benefits of building applications as cross-organisational compositions of Web services and have explained how middleware facilitates their development. We have characterised the resulting service compositions as cross-domain, middleware-based distributed software systems and discussed how the complexity of the operating environment they inhabit leads to an increased propensity for failures. The focus of our work is on a severe type of failures that we refer to as system-level failures. We provided a detailed model of their causes, consequences, temporal and propagation characteristics. Our review of existing testing and fault tolerance approaches concluded that there are no techniques that can remove or tolerate all faults leading to system-level failures. Consequently, human system operators have to handle their occurrence at runtime. We have conveyed a sense of the challenges the detection and diagnosis of system-level failures confront operators with and explained the factors making the common approach difficult, time-consuming and error-prone.

This analysis motivates the goal of our thesis to investigate how system operators can be enabled to reduce the severity of outages through improved failure management capabilities. In particular, we aim to automate failure detection and assist system operators to perform more timely diagnoses of system-level failures by offering access to a largely untapped reservoir of data about system structure and operation. Next, we discuss how we obtain this data through comprehensive and largely non-intrusive monitoring at all layers of a service compositions.

Chapter 3

Monitoring

In this chapter we provide a conceptual overview of our approach to monitor the operation of a service composition at all its layers. After explaining the motivation for this work and identifying constraints and requirements on such a monitoring approach, we present the available sources of data at the various layers and the types of metrics they produce. We discuss three types of measurement approaches through which we can observe these metrics non-intrusively. Then, we explain how such monitoring supports the data-driven approach to detection and diagnosis that we are aiming for. We close this chapter with a review of existing work on monitoring in order to discuss how our approach differs from and builds on those.

We provide an overview of a prototype monitoring framework called Monere that implements the concepts and requirements discussed in this chapter in Section 5.1.

3.1 Motivation

Industrial processing plants, such as oil refineries, chemical processing and power plants, are complex systems and this complexity needs to be managed. The implementation of these processes relies on a large number of components that have been manufactured by different providers. These components are often used as black boxes that perform some operations on their inputs and produce results that are then fed to other components. There are dependencies between the various components and subsystems of a large plant. In order to tackle this complexity, some form of industrial process control is usually applied. This introduces a framework of operator guidelines and tools to support a relatively small number of operators with managing a complex process that is distributed across a large number of heterogeneous components. An automated control loop is often at the core of industrial process control. The control loop monitors and adjusts the operation of components so as to maintain liveness and safety of the overall process. For example, the rotations per minute of a centrifuge in a nuclear processing plant need to be maintained within a safe operating range in order to prevent damage to expensive equipment and catastrophic malfunctions. In case an issue cannot be resolved autonomously, operators are alerted and then use an infrastructure of sensors and actuators to identify and resolve the problem. The basis for this process control loop is rigorous data collection about components and processes performed by numerous sensors. The measurements made by these sensors provide actionable information that is used to decide

on appropriate control actions. Monitoring is a key component of industrial process control.

The failure management capabilities of distributed software systems lack much of the sophistication of industrial control mechanisms. In practice, less rigorous processes may suffice for service compositions as their failures are unlikely to have catastrophic consequences such as outages of industrial plants may entail. Nevertheless, as we have illustrated in the previous chapter, system-level failures at runtime can cause lengthy and wide-ranging outages. In order to reduce their harmful effects, it is necessary to handle their occurrence more efficiently.

Our goal is to support human system operators with the management of complex systems, even if on a smaller scale and in a more limited manner than is typical for industrial process control scenarios. We want to improve failure management by enabling a more principled approach to detection and diagnosis. Such a principled approach is in turn based on monitoring the operation of a system in order to aid system operators in understanding it, especially its behaviour under failure conditions.

In order to accurately detect and successfully diagnose a system-level failure at runtime, it may be necessary to evaluate a variety of conditions. For software systems we do not posses detailed catalogues of symptoms and failure conditions. This makes it difficult to determine sets of metrics that are both necessary and sufficient to guarantee our ability to detect and diagnose specific failures. Therefore, our approach exploits data that is produced as a by-product of system operation. Most components produce some data about their activities, which can often be measured in some way. We measure a wide variety of metrics about the operation of an application and its underlying components.

The idea is to make information explicit that is otherwise hidden. Without such a monitoring facility there is only a large set of implicitly available data that is distributed across an entire system. This data is spread across layers, hosts and even different administrative domains. It is available from a large variety of different sources, such as the operating system, Web and application servers, Web service runtimes, etc. and is accessible through different means. Our approach measures, integrates and correlates all this data to make it available to system operators.

Comprehensive monitoring of the operation of a service composition at all its layers allows us to maintain historic records of measurements in a central repository. Access to such records can help operators with determining what might have contributed to a failure even after the fact and makes it feasible to distinguish between values of metrics during normal operation and values obtained under failure conditions. It enables operators to inspect the development of metrics over time in response to application activity and spot harmful trends and anomalies. It allows us to find out whether monitoring of this kind can significantly reduce the downtime from failures and to determine any benefits of supporting system operators with data about the structure and operation of a service composition.

Monitoring cross-domain, middleware-based service compositions needs to satisfy a few basic requirements. An important requirement is to obtain measurements in a non-intrusive manner. In the context of monitoring, *intrusiveness* can be defined along three dimensions. The one we are primarily concerned with is the degree to which instrumentation requires significant changes to the monitored system. Instrumentation should not necessitate substantial changes to the monitored system, such as source

code modifications or extensive changes to installed operating system packages. In the context of service compositions this is an important requirement. The software components comprising a service composition have generally not been designed and developed to facilitate their instrumentation. There are no standard instrumentation interfaces used by a broad set of components that would enable an observer to easily collect a wide variety of measurements. Approaches that rely on modification to components in order to instrument them are not suitable as we cannot assume access to their source code and in some cases do not even have control over relevant components that reside in separate administrative domains. Another alternative to retrofit legacy systems with monitoring capabilities is to monitor events as they occur in the operating system kernel. However, the metrics this produces describe kernel-level events, such as cache accesses and page allocations. There is a conceptual gap between such metrics and the behaviour at higher levels that gave cause to them. An approach that can offer higher-level metrics while still avoiding intrusive instrumentation would be preferable.

Another dimension of intrusiveness is the performance overhead introduced by the monitoring infrastructure. For example, execution of the monitoring logic itself consumes computational resources, such as CPU cycles, memory and disk space. Measurements are obtained from components distributed and accessible across a network and monitoring necessarily uses some of the available network bandwidth. Finally, the actions performed to take measurements may need to interact with the monitored component and thus impose an overhead on its operation. These costs should ideally be less than any benefits we derive from monitoring. The final dimension of intrusiveness is the degree of perturbance caused to the monitored system. Such interference with its normal operation is a particularly relevant concern for real-time systems, which we do not consider in this work.

The *integration and correlation* of measurements form another key requirement. We have already discussed the need to integrate data collected from among the components of a distributed system. In addition to this, operators often want to determine how a system behaved in response to application activity. Given the large number of different components and their measurements, it might be useful to determine relationships between different measurements. For example, it can often be helpful to be able to correlate application activity at a particular point in time with other measurements and events that occurred at lower layers in the system.

Taken together, this and the next chapter provide a conceptual overview of our approach to support data-driven detection and diagnosis of system-level failures based on making information about system structure and behaviour explicit. Our approaches to monitoring and dependency discovery do not necessarily propose novel techniques, but instead they reuse, combine and in some instances modify existing techniques. However, their discussion in this and the next chapter as well as the brief description of their implementations in the main evaluation chapters forms an important part of the contribution of this thesis for two main reasons. First, our approach represents the first one to provide for the comprehensive monitoring of a service composition. It is the first one to monitor a service composition at all its layers, discover all relevant components underlying a composition and that does so in a non-intrusive manner. There is no literature that describes a similar non-proprietary monitoring system. Therefore,

this description of the concepts our approach is based on and a review of how these can be made to work in practice is likely to be useful to others. Second, the main aim of this thesis is to investigate the effects of monitoring and dependency discovery on the detection and diagnosis of system-level failures. There is no existing system that we could have referenced and that would have allowed us to perform the corresponding experiments necessary to evaluate this question. It is thus important to clearly define and understand the constructs whose effect we are actually measuring in our experiments.

3.2 System Layers and Data Sources

Each request that is received by a Web service is processed by numerous components at the service, middleware and operating system layers. Not all components that are present are involved in processing every single request, but many of them are. As we have already discussed, a request usually traverses all three layers. Failures can develop from the behaviour of components at any of these layers. In order to increase the likelihood of observing conditions that actually lead to failures, it is necessary to monitor at all layers supporting a Web service.

The service layer is comprised of services, which act as interfaces to units of application logic. In the case of service compositions these often take the form of Web service interfaces and BPEL processes. The components at this level constitute the part of an application which is visible across the Internet and that can be invoked by clients. The only aspects that are directly observable at this level are the presence of a service and its interface description. All other aspects of the behaviour of service-level components need to be obtained by indirect observation at their respective underlying middleware components.

The components at the middleware level represent the implementation of a service. These components are involved in processing a request beginning with its initial receipt and all the way through until processing is completed. This layer consists of different components (e.g., application server, database management system) that vary in the kinds of functionality they provide. Their functionality may in turn define what kinds of behaviour need to be observed. For example, to determine the health or performance of a Grid job scheduler requires a different set of metrics than determining the health of a file system or the responsiveness of an application server. In some cases, measurements can be taken directly from the components under observation. Other components need to be monitored by components containing or hosting them, such as for example an application server.

Every operation needs to use at least some of the features of the operating system and will eventually have some effect at this level. The operating system layer consists of components such as network interfaces, file systems and of resources such as CPU time and memory. It offers abstractions that provide access to these resources (e.g., threads, sockets). Most operating systems provide some mechanisms to access data about several aspects of their operation (e.g., CPU time, memory usage, statistics about network interfaces, etc.) and allow for the collection of indirect measurements of most components executing at the higher levels. Next, we review the actual types of metrics that can be measured at these layers and explain how they can be collected non-intrusively.

3.3 Metrics

There is a large number of metrics that can be obtained from monitoring a service composition and its underlying components. We use the term *metric* to refer to a representation of an attribute of some aspect of the behaviour of a component that we can observe and measure. *Measurement* is the process of quantifying the state of an attribute as it was observed at a particular point in time. Values are assigned from a range of possible values as defined by a unit of measurement. For example, latency can be regarded as a metric that measures the time it takes for a network packet to travel between two endpoints. It can be measured in milliseconds, which specifies a range consisting of positive real numbers. Measurement can encompass both the observation of direct metrics whose value is recorded as it is observed and indirect ones, which are calculated based on several other metrics (e.g., a metric that describes a proportion or rate). Each of the components comprising a service composition has several attributes that can be monitored and that may be useful to detect and diagnose the occurrence of system-level failures. The set of metrics we measure in our approach can be divided into a number of categories depending on the aspect of system operation that they capture. Figure 3.1 shows an overview of the types of metrics we measure at the different layers of the system.

Layer / Metric	Operating System	Middleware	Service
Availability	●	●	●
Events & Activity	●	●	●
Performance	●	●	●
Resource Usage	●	●	

Figure 3.1: The types of metrics available at the three main layers of a service composition.

3.3.1 Availability

A symptom shared by some failures is the unavailability of components. [Avizienis et al., 2004] defines availability as the "*readiness for correct service*". However, in order to determine whether a component is functioning correctly often requires application-specific knowledge to analyse responses. Instead, our *availability* metric measures the time a component has been ready to provide service. We consider a component to be available at any given moment in time, if it responds to incoming requests or consumes computational resources. In addition to the most recent status of each component, availability metrics also include statistics about past availability, such as the average time a component was available between failures and its total and average downtime per failure. Availability can be measured for most components, including Web services, middleware components as well as operating system processes and components. In most cases it is measured locally, but for components that can be invoked across a network, it is necessary to also measure availability as it is perceived by remote clients. In these scenarios network connectivity issues or a firewall blocking some traffic may limit the availability experienced by

some clients, even though the service itself has been up and running continuously.

Maintaining records of the availability of components is useful as it can often serve as a first indication that a failure has occurred. In combination with a dependency graph, this information can even help to eliminate irrelevant components from the diagnosis process. For example, if a number of unavailable components all share a dependency on another component that is also unavailable, this may point to a possible root component for whatever the current failure is. An operator can then first of all focus her investigation on this component and omit all other branches of dependencies from immediate consideration.

3.3.2 Exceptional Events and Application Activity

Another type of metric that can potentially support failure diagnosis is information about events of interest that occur anywhere in the system. We distinguish *exceptional events* from regular *application activity*. The former represents failures experienced by a component and the latter events that signify normal activities performed by an application. The metric is in the form of information about an event as it may have been recorded in error and warning messages in a log file or published through some other means. The captured data includes at least the time of occurrence, the location to identify the host and component that logged it and the type of the detected event. It can include additional information that further describes the event. Operating systems usually maintain a variety of log files and components within an application sever often share its log file to record noteworthy events. A middleware component may record an exception it has experienced due to having run out of memory or repeatedly being unable to reach another component. A firewall log can contain data about incoming network traffic that has been blocked.

Application activity is not generally recorded in a consistent manner. Instead, most applications will only log a few checkpoints or milestones in their execution, such as their successful initialisation at startup. The situation is different for applications expressed using BPEL. A BPEL engine usually records each action and major state change of the process instances running within it. Recorded activities include events such as the receipt of a message by a BPEL process, the invocation of partner services or operations performed on the data stored within a process instance.

Exceptional and application events may be useful metrics for the diagnosis of system-level failures for a number of reasons. First, they provide operators with an overview of exceptional events, at least some of which are likely to have contributed to a failure. The information associated with these events may even accurately describe the circumstances that have led to the event or describe the nature of an error. Second, the value of being able to pinpoint what action the application performed at a particular point in time stems from the ability to correlate this information with other measurements taken at the system level. This can provide insight into what behaviour the application caused at the lower levels.

3.3.3 Performance Indicators

The performance of a computer system or of any of its components can be described as a measure of the amount of work it carries out within a span of time. *Performance indicators* provide a sense of the speed or the rate of progress at which certain activities take place. Performance metrics are often either

expressed as an amount of time an activity takes to complete or as a rate such as a count of the number of events or steps performed per unit of time. Similarly, a performance metric can also describe the quantity of something processed or transmitted per unit of time (e.g., bandwidth in bits per second). Performance metrics can be obtained for many different components at all layers. For a Web service it is possible to observe and calculate the average execution time it has achieved for requests within a given time period. Given the number of requests it has received within that time, we can calculate its throughput (i.e. the number of requests it completed successfully within a unit of time). Performance metrics of a network interface may include the number of bytes that have been sent and received per minute over the past hour and the number or rate of dropped packets. Another example of performance metrics are the average query execution times achieved by an object-relational mapping tool, the amount of data it reads from or writes to the database within an hour and the number and proportion of failed transactions out of all transactions performed.

Performance metrics can highlight components whose performance has degraded considerably and that may contribute to some failures. They are useful for identifying components that experience high load, such as for example a Web service that has started to become overloaded by concurrent requests, which leads to an increased response time that can cause timeouts to occur at its clients.

3.3.4 Resource Usage Metrics

A resource is the supply of any physical or virtual component that is used for the operation of a computer system, such as the processing of requests. For example, a physical resource is the available capacity of a storage device for storing bits and any volumes or files are virtual resources based on this physical one. Other examples of resources are memory, network connections, CPU time, operating system threads and processes. *Resource usage metrics* provide a measure of the consumption of various computational resources by components and are typically given as amounts or proportions. Various resource usage metrics are available from components at all layers. For example, an OS process makes use of a number of threads to execute its operations and file descriptors to perform socket communication. For a Java process executing a JVM we can determine the CPU time and memory it is consuming, the number of classes that have been loaded by the JVM and the amount of heap space used. The number of currently idle and busy compute nodes is one resource metric of Grid clusters.

Resource usage metrics can facilitate failure detection and diagnosis in a number of ways. They support the identification of components that have exceeded certain limits and can also point out deficiencies in the configuration of components where a demanding workload may result in resource exhaustion. For example, we may find that the OS needs to grant a higher number of file descriptors to processes in order to support a particular workload. Or measurements of thread usage may indicate that the thread pool size of two related components needs to be reduced in order not to exceed available resources. Resource usage metrics can also help us determine whether a component has become unresponsive due to having crashed or whether it is still consuming a high and varying amount of computational resources, which is characteristic of a component that operates under heavy load.

3.4 Measurement Approaches

There are several measurement techniques that can be used to observe metrics while at the same time avoiding intrusive modifications that require changes to source code or control over components in separate administrative domains. Most of these techniques implement one of three general approaches to measurement. The three approaches that we use are event tracing, interception and sampling. Figure 3.2 shows an overview of the types of metrics that we measure in a service composition using these approaches.

Approach / Metric	Event Tracing	Interception	Sampling
Availability			●
Events & Activity	●	●	
Performance		●	●
Resource Usage			●

Figure 3.2: Overview of the metric types obtainable through three non-intrusive measurement approaches.

Event tracing describes the observation of a component for the occurrence of an event of interest, such as an error message or a state change performed by an application and the subsequent recording of information about the detected event. Events are captured in the form of event metrics as we have discussed them in the previous section. In middleware-based systems this pattern can often be implemented in a non-intrusive manner as many components record noteworthy events in log files or offer notifications about events to interested subscribers via publish-subscribe APIs. The types of events we consider are fairly generic and common ones (i.e. errors, warnings, state changes, invocations) and as such are available from many components. Some components even allow the amount of detail committed to log files to be configured. With event tracing, data is recorded whenever an event of interest occurs and therefore no event is missed. The overhead of event racing is primarily dependent on the frequency with which events of interest occur.

Interception of requests is another measurement approach we use. Interception works by inserting a measurement probe between two endpoints that communicate with each other by message passing. The probe acts as a proxy that intercepts the communication and is thus able to observe several attributes that characterise the interaction between endpoints. Interception is well-suited to obtain performance indicators by observation of the communication occurring between Web services. An interceptor can analyse incoming and outgoing pairs of messages in order to derive response times, execution times, latencies, counts of failed requests and so on. Interception can also make events observable that are not usually logged by any component, such as the sending of requests and receipt of response messages, and to calculate metrics based on the occurrence of these events. Except for a usually small overhead

from the interception and processing of a message, the introduction of a proxy in the form of a message handler can be transparent to both endpoints. Enterprise middleware frameworks often allow for message handlers to be inserted into existing message processing chains through modification of configuration files without the need to change the monitored services in any way. Similar to what is the case with event tracing, each receipt and sending of a message can be recorded and the overhead of this method is directly proportional to the number of messages exchanged between monitored endpoints.

The third and final pattern that we use to obtain measurements in a non-intrusive manner is *sampling*. Sampling applies to a variety of different implementations that all work on the basis of some sort of sensor taking measurements at intervals that are usually fixed. Most sensors are specialised to measure certain types of attributes from particular types of components and periodically poll these components to perform their measurements. Sampling can be used for a variety of metrics. We rely on it to obtain resource usage and some performance metrics as well as to determine the availability of components. The data produced by a sensor can just be a simple count or may be more complex than that. Our requirement to avoid intrusive modifications necessarily limits the kinds of data that can be obtained as many components have not been designed to be monitored. However, sampling is still suitable as a non-intrusive measurement approach. For example, a sampling operation can consist of reading values in the process information pseudo-file system[1] about the resource usage of processes or invoke middleware components and services on mock requests to observe their behaviour. There is a great deal of flexibility in how to implement a sampling strategy. Unlike with the first two approaches, measurements based on sampling do not necessarily capture each and every change of a monitored attribute. The overhead incurred is a function of the sampling frequency used. We will review examples of techniques for such sensors when we discuss the implementation of our prototype.

There is a trade-off between the sampling frequency, the overhead incurred from sampling and the amount of useful information we gain from each measurement. It is probably wasteful to measure the availability of a Web service once per second. This incurs a substantial overhead even though the availability of the service is unlikely to change within any one-second interval. If we measure the memory usage of an OS process, the situation is different as the corresponding state can change quickly. A memory leak usually causes the amount of memory used by the process to oscillate somewhat between local peaks and lows and displays a trend of steadily increasing memory usage over time. If we only sample the amount of used memory every few minutes, we may fail to record parts of this pattern at inopportune moments in its development, which can make it more difficult for an operator to identify the cause of a failure. In this case, it would be more reasonable to take measurements at least every few seconds. Finally, consider a metric that describes the amount of free space available in a file system. A suitable sampling frequency may be dependent on the goal we pursue with monitoring. If the goal of monitoring this metric is to assist with the detection and diagnosis of failure causes, then the rate at which the file system space is used is less important than whether there is a low, medium or high amount of free space left. A relatively low sampling frequency may be justified in this case. However, if the goal was to ensure

[1] The process information pseudo-file system provides access to many kernel data structures of a Linux operating system.

that sufficient storage space was provisioned at all times, then more frequent measurements to determine the rate at which space gets used would be necessary.

It is important to choose an appropriate sampling frequency. If we measure a metric too frequently, then we incur a high overhead, but are not likely to gain much useful information between measurements. On the other hand, a low sampling frequency saves some of the overhead, but may result in missing important changes in the state of what is being measured. Apart from an acceptable overhead, as the examples above illustrate, the choice of a suitable or even optimal sampling frequency depends on a number of other factors and hence needs to be determined separately for each metric. It depends on how quickly the thing being measured can change as well as how quickly it does change in practice. This governs how much information can be gained between any two measurements. It also depends on how much information is necessary to support the goal for which monitoring is applied (i.e. failure detection and diagnosis). We do not address how to dynamically establish optimal sampling frequencies at runtime. Instead, in our prototype we have chosen collection intervals for each metric based on experience and trial and error.

3.5 Management Information Base

What does monitoring of the kind we have described change about the problem of system-level failure detection and diagnosis? Monitoring primarily achieves the integration of data about system operation. It collects and provides access to data that is otherwise scattered across components and hosts. Without monitoring in place operators would have to access this data using a variety of mechanisms. Additionally, in some cases the integration encompasses data whose measurement requires the preparation of additional instrumentation and that would not be available without it. Essentially, the result of applying a monitoring approach such as ours is to produce a Management Information Base (MIB), similar to databases of network assets as they are used for network management. The MIB consists of historic and up-to-date measurements for multiple metrics as they can be measured within an administrative domain. That is, each administrative domain using an approach such as ours would end up with its own MIB that contains measurements for components within that domain as well as for certain metrics of service-level dependencies across domain boundaries.

The availability of an MIB improves on the situation that we have outlined in Chapter 2 in a number of ways. First, an MIB could relieve operators of a diagnosis process that often involves iterations of trial-and-error. Typically, an operator searches the different parts of the system for relevant data and may discover that the data needed to test an hypothesis about the failure cause is not available. After having installed additional monitoring to capture these relevant metrics, the operator may need to re-execute the application and wait for recurrence of the problem. Only then is she able to test her assumption and may determine that it is indeed necessary to monitor even more aspects of the system. This diagnosis by trial-and-error can become time-consuming. Instead, an MIB provides a relatively complete overview of system behaviour expressed through a broad set of metrics extracted from numerous components that are relevant to a service composition. The MIB enables operators to perform queries over this data and allows us to aggregate and visualise it in different ways. More specialised debugging tools can be

employed should this be necessary, but diagnosis is now primarily based on the data collected in the MIB.

Furthermore, the availability of historic measurements for most metrics provides an overview of their continuous development over time, which may help operators to spot potential anomalies and other relevant trends. Another advantage of an MIB is that it enables the correlation of the large number of measurements stored in this repository. Correlation based on the timestamps of measurements is a simple way to achieve this. Even though timestamp-based correlation cannot guarantee to establish causality, it is still useful to determine relationships among events in the system. It can reveal how components at the middleware and OS layer have reacted to specific application activity. Finally, an MIB enables us to use such data in various kinds of automated analyses. We focus on how such an MIB can be exploited in order to automate the detection of failures. All this appears to represent an improvement when compared with the status-quo.

3.6 Related Work

We compare our efforts with related work that falls into three categories; the monitoring of service-oriented computing systems, the monitoring of distributed systems and some commercial monitoring tools.

3.6.1 Monitoring of Service-Oriented Systems

BPEL service compositions can form complex invocation sequences among partner services. In [De Pauw et al., 2005], the authors present the WS Navigator tool to visualise these interactions. WS Navigator records various events as they are emitted by BPEL engines during the execution of a BPEL composition (e.g., process start and finish, partner service invocation, delivery of response message). A data collector component within the WebSphere Application Server [IBM, 2012] records several metrics in log files, such as network delays, network transit times of messages, message size and content, server processing times, endpoint references of partner services and so on. The individual events are correlated into corresponding transactions between Web services and visualised in a number of ways. WS Navigator allows system operators to inspect the exchanged messages, view a high-level topology of the services that have been involved in exchanging these messages and view sequence diagrams of the interactions between partner services. WS Navigator exploits data that is readily available from most BPEL engines.

The VieDAME framework [Moser et al., 2008] addresses two shortcomings of BPEL service compositions. The first one, which we do not consider in more detail here, is the lack of support for dynamic replacement of partner services at runtime. The second shortcoming is the lack of support for detailed monitoring of BPEL process execution. VieDAME monitors several Quality of Service (QoS) attributes of a BPEL service composition. It integrates an Interception and Adaptation Layer (IAL) into a BPEL engine to intercept messages to and from partner services. A monitoring component then inspects all intercepted messages and stores measurements for each partner service in a database. It measures all partner service invocations and calculates the average response time, whether or not an invocation was

successful, the availability of the partner service and its accuracy ($1 - failed\_requests/total\_requests$).

An extension to VieDAME is presented in [Moser et al., 2010]. Users can express event processing rules to filter and query events. For example, a user can query the response times of a particular partner service or instruct the monitoring runtime to record when certain messages have been outstanding for a specific period of time. The authors recognise that many components outside of a BPEL engine may contain relevant information about the execution of a service composition. However, the paper does not address how these additional components should be monitored and only allows for the packaging and submission of existing measurements via adapters to the existing monitoring runtime.

The approach described in [Wetzstein et al., 2009] aims to determine why certain Key Performance Indicators (KPI) of a service composition have been violated. A KPI is typically a high-level objective, such as for example that the fulfilment times for orders should remain under three working days. Users specify which metrics should be monitored for the evaluation of a particular KPI. These metrics can consist of events that a BPEL engine produces and of QoS metrics, such as the availability of Web services and their response times. The QoS metrics are measured by a component that periodically polls the BPEL process and its partner services. The measurements are persisted in a database and are then used to learn decision trees that classify whether or not a KPI has been violated based on the values of its relevant metrics. This can reveal which factors are most important for the achievement and violation of a KPI. For example, a decision tree might show that a response time value above a certain threshold does consistently lead to the violation of a KPI.

In [Guinea et al., 2011] the authors recognise that monitoring purely at the service-level may not be sufficient to enable correct adaptations at runtime and propose to also monitor the underlying infrastructure level. By this they refer to an Infrastructure-as-a-Service (IaaS) service, such as the compute nodes provided by a cloud computing provider. The approach monitors the execution of a BPEL process by polling it periodically to obtain reliability metrics and average response time measurements. At the infrastructure layer, the approach records, if an invocation was successful or whether it resulted in some kind of error message. Users can also define domain-specific metrics. For example, the KPI might state that an entire business process should not exceed 60 minutes. To evaluate this KPI, the approach would measure the duration of the entire process and of each process activity at the service level and record which node was used at the infrastructure level and whether the execution on that node was successful. As in [Wetzstein et al., 2009], the authors then determine the influential factors for the outcome to determine whether adaptation is needed at the service or at the infrastructure level.

Another approach that aims to dynamically adapt Web service compositions at runtime in order to improve their dependability is presented in [Chen, 2008]. The WS-Mediator approach relies on a network of so-called Sub-Mediators (SM) that are installed at different geographic locations (e.g., at the clients) so as to monitor Web services from different vantage points. An SM measures various dependability attributes of a Web service by periodically executing test scripts that exercise its operations. It measures the availability, round trip response times and any types of failures that occur. This allows for the calculation of the recent dependability characteristics of a Web service. WS-Mediator uses this data

in order to implement resilient-explicit reconfiguration. Clients specify the dependability criteria that they deem important and WS-Mediator then sorts the available partner services accordingly. It can then invoke the service with the best score. It also employs several fault tolerance techniques to mask various failures in these invocations. Apart from retrying an invocation on the next-best alternative service or invoking several of the candidate services in parallel, it can also perform the invocation from another SM that has better connectivity to the target service. WS-Mediator transparently handles these issues and then relays the invocation response to the client.

Work on the monitoring of service-oriented computing systems is primarily concerned with observing the service level. Accordingly, most approaches focus on high-level metrics in addition to a few common QoS metrics as they are used for the evaluation of high-level business objectives (e.g., key performance indicators). Furthermore, most related work is not based on a detailed fault model, which may explain why the approaches do not find it necessary to integrate data from any of the underlying layers.

Our approach to monitoring recognises that the service level is just an additional layer on top of what is essentially a middleware-based distributed system. We integrate measurements at the service level as just one of several relevant layers that need to be observed. Furthermore, we consider a wider set of metrics as it is necessary to detect and diagnose system-level failures. The failures our monitoring approach targets prevent a service composition from making progress and can arise in any of the many components that support its execution, which necessitates the monitoring of these components. In summary, we demonstrate how to monitor a service composition at all its layers, consider metrics for all the components that a service composition depends on for its reliable operation and show how to measure these in a non-intrusive manner.

3.6.2 Monitoring of Distributed Systems

In [Al-Shaer et al., 1999] the authors present the HiFi framework to monitor a large number of events and nodes in a scalable manner. HiFi approaches this goal through hierarchical filtering of monitoring events that can be distributed across a set of monitoring agents. Users specify which primitive and composite events should be monitored. Local monitoring agents (LMA), which are installed on each node, can detect primitive events. If a primitive event matches one of the event filters at an LMA, it is recorded and passed on to a domain monitoring agent (DMA). The DMA is responsible for a set of LMAs and can correlate the events it receives from them into the composite events the basic ones may belong to. Partial composite events can be submitted to even higher-level DMAs, which join partial composite events from several lower-level DMAs and so on. In this way, only events that match a filter are recorded and event processing is distributed across a number of nodes.

Ganglia [Massie et al., 2004] applies hierarchical monitoring to large clusters of compute nodes. Each cluster selects a representative node, which aggregates the measurements of all nodes in its cluster. The representative nodes for a set of federated clusters report these aggregated measurements to a node at the next level up, which can then in turn aggregate the measurements from this set of clusters. Ganglia measures the number of CPUs, CPU clock speeds, CPU and memory utilisation, the number of running processes and swap space statistics. The authors find that hierarchical aggregation in this manner leads

to approximately linear scalability of memory usage and communication overhead with the number of nodes being monitored across all clusters.

Chopstix [Bhatia et al., 2008] deals with how to monitor and record high-frequency events as they occur in operating system kernels in an efficient manner. Chopstix instruments an operating system kernel to measure page allocations, process scheduling events, locking, input/output operations, L2-cache utilisation, system call events and so on. All of these metrics produce a large number of events and the goal is to avoid missing too many of their occurrences. The main contribution of Chopstix is a probabilistic data structure that can track a large number of events in a more efficient manner than is possible with a deterministic approach. This data structure adapts its sampling rate for an event as a decreasing function of this event's recent population within a monitoring interval. This approach allows the Chopstix to achieve 99.9% coverage of events of interest at minimal CPU overheads.

Magpie [Barham et al., 2004] is a framework for obtaining accurate workload models by monitoring the resource usage along request processing paths. In contrast to the approach presented in [Chen et al., 2002], Magpie does not need to instrument middleware components in order to maintain and propagate request identifiers across components. Instead, Magpie relies on expert users to provide request schemas that enable it to tie various monitored events together into their corresponding requests. These schemas specify how request processing in various components results in different basic events and how the events for a particular event can be related to each other. For example, some component that begins processing a request assigns an internal identifier to it, but also notes a particular thread identifier. This same thread identifier may also be used in events that denote the beginning and end of processing the request in a particular application server. This allows Magpie to identify that these three events belong to the same request and join them together. Magpie instruments the Windows NT kernel and modifies a packet capture library in order to monitor an application and middleware at all the points at which request processing transfers between components. The end result are detailed resource consumption measurements for requests.

E2EProf [Agarwala et al., 2007] supports the identification of performance bottlenecks in distributed systems by discovering request processing paths and the associated processing latencies that each component along the path contributes. E2EProf does not need access to an application's source code, but instead uses kernel level network traces to collect timestamps for every inter-component message along with information to identify the endpoints of each exchange. These traces are then converted into time-series data streams and an algorithm is applied to discover likely causal pairs of components. It does so by computing cross-correlations for the time-series signals. Cross-correlation analysis is used in digital signal processing to discover similarities between signals. The resulting request paths are updated periodically to reveal up-to-date inter-component interactions involved in processing requests and to reveal how much latency each component contributes.

Two approaches that monitor exclusively at the kernel level are DTrace [Cantrill et al., 2004] and SysProf [Agarwala and Schwan, 2006]. DTrace allows users to specify instrumentation probes as scripts written in a subset of the C programming language. Probes fire when certain events occur and can make

use of various kernel level modules that implement different kinds of instrumentation. For example, a probe can be instructed to activate itself when a process is created or destroyed and it could use an instrumentation to measure the time during which the process was active. Measurements can be filtered and aggregated in the kernel, before being returned to the requesting user-level appliaction. DTrace thus enables users to query operating system kernels for a wide variety of information.

Similarly, SysProf [Agarwala and Schwan, 2006] implements a kernel module called Kprof, which receives data from an instrumented kernel. Kprof offers user-level programs an API through which to instruct it to collect data about various events, such as scheduling events, system call events, network events or file system events. The resource consumption of request processing can be measured using these types of events. An analysis component correlates the monitoring data from Kprof and aggregates it into higher-level metrics, such as CPU usage, throughput and the identity of request-response pairs among messages.

PlanetSeer [Zhang et al., 2004] is a monitoring framework that can detect network path anomalies in wide-area communication among a large number of nodes. Its main contribution is the combination of passive and active probing in order to improve the scalability of its measurement operations. Low overhead probing is performed continuously on all nodes in the system and more intrusive probing is only activated, once a problem is suspected. Passive probing relies on tcpdump [tcp, 2012] in order to monitor packet traffic, identify message paths, measure timeouts, record sequences numbers, retransmissions and round trip times. Simple heuristics are applied to detect possible failures. For example, the observation of a number of consecutive timeouts along a path, indicates that data packets or their acknowledgements have not been delivered and triggers active probing. Active probing is based on issuing traceroutes from a number of geographically distributed nodes in order to observe the path in question in more detail. This allows to collect information about message routes along with transit delays along each hop.

Related work on monitoring of distributed systems proposes techniques that address different aspects of the monitoring problem space. For example, some approaches enable the monitoring of legacy applications by instrumenting the operating system kernel while others propose techniques to make the monitoring of a large number of nodes and of high-frequency events more scalable. The evaluation of these techniques is generally an intrinsic one that demonstrates its feasibility and in some cases quantifies its performance characteristics.

Our monitoring approach does not propose any new techniques. However, its value derives from the fact that it supports our evidence-based determination of the effects of such a data-driven approach. We demonstrate what kinds of metrics can be measured non-intrusively in middleware-based systems and determine in how far this data can support failure detection and diagnosis. A determination of the impact of a monitoring approach on the systems management activities that it should support is usually lacking from most existing work on monitoring.

3.6.3 Commercial Monitoring Frameworks

There are several large commercial monitoring frameworks, such as HP Network Management Center [hp:, 2012] and IBM Tivoli [ibm, 2012]. These monitoring frameworks are comprised of numerous

modules that deal with various aspects of network and systems management. They support the discovery of network devices and Tivoli also discovers dependencies of server and mainframe applications on storage and network devices. They maintain databases of assets and support change management. They can monitor the availability and performance of many different components and each provide some form of automated diagnostics. Monitoring is typically performed by agents that are deployed on individual hosts and that communicate with a set of monitoring servers. In addition, commercial monitoring frameworks often offer some kind of data warehousing which manages the storage of a large amount of device information and monitoring data and makes this data available for inspection, analysis and for further processing.

These frameworks provide extensive functionality, but they are proprietary systems. This makes it difficult to ascertain in detail what their exact capabilities are and how they have been implemented beyond the information that is published in marketing materials. Furthermore, they are often designed to work with proprietary components whose behaviour is well-understood. Research on monitoring typically involves a substantial implementation effort. It might be helpful, if academia could gain access to some of the functionality that is offered by these commercial monitoring frameworks. Otherwise, it can be difficult to compare our efforts fairly with the techniques implemented by commercial frameworks.

One of the key requirements that we have identified for the monitoring of service compositions is to collect only data that is readily available[2] without necessitating any intrusive changes. Splunk [Splunk Inc., 2012] is a commercial monitoring system that is close to this philosophy. It collects a wide variety of readily available data, indexes it and makes it searchable. An administrator can then issue keyword-based searches against the information collected from a number of log files. In its early incarnations, Splunk was mainly a log file processor, but many additional data sources have been added over the years.

Our prototype monitoring framework, Monere, which implements the requirements and concepts discussed in this chapter and of which we give an overview in Section 5.1, differs from Splunk in a number of ways. Splunk's main goal is to index large amounts of measurement data to make it searchable. The main objective of our monitoring framework is to support data-driven detection and diagnosis. Our framework can therefore avoid any overhead from indexing a large amount of data to improve its searchability. Our primary concern was to determine what types of metrics and which aspects of the behaviour of a service composition can be measured non-intrusively. Furthermore, an important feature of Monere is its integration of dependency information in order to support system operators during diagnosis. Our framework also addresses the need to exchange measurements across administrative domain boundaries in way that is scalable and that takes the potential for trust issues between service providers and clients into account. In Monere, visibility of components in other domains is restricted to the service level and providers maintain control over which metrics they want to publish to clients. We discuss this feature in more detail in Section 4.5.

Apart from these differences, we were unable to use Splunk as the monitoring backend for our

[2]This does not mean that their measurement is simple to implement.

work for a number of reasons. First, we had to ensure that we could monitor all relevant components of a service composition and did not want to have to wait for a third party to eventually integrate support for these components into their product. Second, Splunk is not open-source, which could have prevented us from examining and describing the type of monitoring that underlies our data-driven approach in detail.

Nagios [nag, 2012] is an open-source monitoring framework that offers functionality to monitor multiple hosts and the service that are deployed on them (e.g., SMTP, HTTP server). We have chosen to extend RHQ [Red Hat, Inc., 2012] instead of Nagios due to two differences. First, RHQ provides better integration of the Java enterprise stack of software components out of the box. Second, its per-component model offers meta information that is missing from Nagios (e.g., the type of a metrics and its units of measurement). One other aspect in which our prototype, Monere, differs from Nagios is that Monere performs component and dependency discovery whereas Nagios requires all components to be configured manually.

Conclusions

In this chapter we have given an overview of how a service composition can be monitored at all its layers. We outlined the key requirements such a monitoring approach needs to cater for, such as not requiring intrusive modifications to the monitored system, imposing a performance overhead that is small in relation to any benefits monitoring may help to realise and the ability to integrate and correlate measurements from different sources. We discussed which parts of a service composition should be monitored and proposed a few types of metrics to monitor for. A brief rationale was provided for calculating the availability of components, listening for events of interest and measuring resource usage and performance metrics. Next, we presented three different measurement approaches that allow us to observe these metrics non-intrusively in most middleware-based service compositions. These approaches are event tracing, interception of requests and sampling. With regard to the latter, we have discussed the trade-offs involved in choosing appropriate collection intervals for metrics. We have explained how the data resulting from such monitoring forms a Management Information Base of measurements and how this MIB can be exploited in order to support system operators during diagnosis and to enable us to begin to automate aspects of failure management. Finally, we have reviewed existing academic and commercial approaches to monitoring service compositions and distributed systems in general in order to explain what our work has to offer in addition to existing contributions.

One of the goals of our work is to help system operators with finding likely causes of failures during diagnosis. And monitoring addresses this goal to some extent by making more information available about the behaviour of a service composition. However, it still confronts system operators with a large unstructured space of data that is the result of the numerous components and the dependencies they form on each other. It could be potentially useful to provide operators with a model of the structure of a service composition and its underlying infrastructure. In the next chapter, we discuss such a model in more detail and explain how the features of middleware-based systems in general and service compositions in particular can be exploited in order to instantiate such dependency graphs automatically.

Chapter 4

Dependency Discovery

In this chapter we motivate the use of dependency graphs as a potentially useful abstraction to support a more principled diagnosis process and describe their structure and how they can be obtained for service compositions in an automated manner. We exploit a characteristic that holds for many middleware-based systems. Namely, that in order to allow the various components they are typically comprised of to work with each other, middleware-based systems often reify information about their deployment. It is possible to process this information to discover much about the structure of a given deployment.

Ater explaining the benefits of dependency graphs, we present the model on which our dependency graphs are based. This model can represent service-level dependencies that extend across administrative domain boundaries and dependencies on and among middleware components down to the level of the operating system. We discuss the various sources of deployment information in middleware-based service compositions and describe a process that exploits these sources for the automated discovery of the components comprising a service composition and the dependencies among them. After explaining how the resulting dependency graphs can support the efficient exchange of measurements between service providers and their clients across administrative domain boundaries, we review existing approaches to dependency discovery and architecture recovery.

4.1 Motivation

A characteristic of applications built as compositions of services is that they depend on a network of services and components. Figure 4.1 shows a simplified overview of the components and dependencies that the system operator of the credit card processor from our example service composition has to navigate during diagnosis. Even though this example omits several details, it illustrates that dependencies routinely span host and domain boundaries, that many components usually depend on several other components and that many dependencies tend to be transitive. In the absence of knowledge about these components and the relationships between them, a system operator faces a difficult problem when looking for failures in large systems.

The large number of components and their interconnectedness, not only at the service-level, but also typically among middleware components, makes it difficult for system operators to maintain an accurate overview of system structure. Documentation in the form of deployment or architecture diagrams, if it

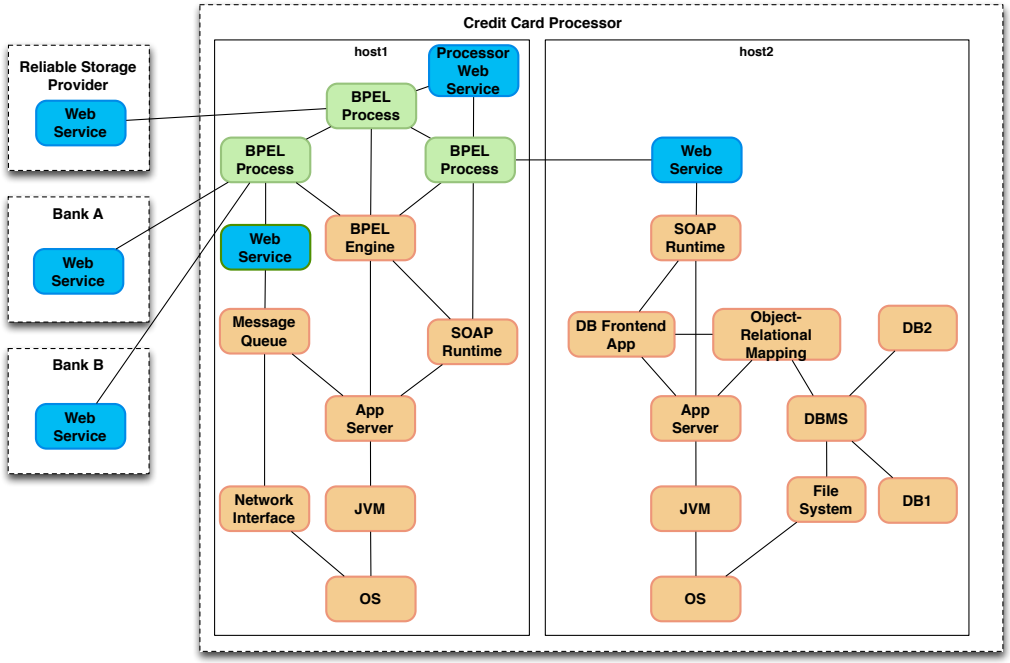


Figure 4.1: Overview of the components and dependencies in the implementation of the credit card processor service from our example service composition.

exists at all, is usually not kept up-to-date as the system structure evolves over time. Often, this information remains implicit within the deployment of a service composition. The problem of maintaining an overview of system structure is exacerbated in case that a system operator is responsible for the maintenance of several different applications. For large deployments it can be challenging to determine which components actually belong to a particular application.

The lack of an accurate overview of system structure makes diagnosis more difficult. It demands of an operator to be intimately familiar with the details of each deployment and in the absence of such familiarity an operator faces several challenges. It is inefficient, to evaluate the metrics of all components one-by-one in the hope of finding valuable clues explaining a failure. However, an operator does not necessarily have much information in order to determine the most promising component to start her investigation or to decide which components to examine next. Another issue is that information about external services, which can contribute to a failure, remain hidden within the specification of a service composition. An operator may need to manually examine a set of BPEL processes in order to collect information on all the Web services it relies on. And finally, without a model of system structure it is difficult to identify components that have no relationship to the ones displaying failure symptoms and that can probably be excluded from the analysis of defects.

In order to address these issues, we want to automatically extract information about system structure from a deployed service composition. Our goal is to discover a per-domain inventory that contains all the components a service composition is actually composed of, shows how these components depend on each other within and across hosts and identifies service-level dependencies within and across domain boundaries. This information can then be made available to system operators during diagnosis in the

form of *structural cross-domain dependency graphs*, which we define in more detail in Section 4.2.

By making the inherent system structure explicit, we may be able to improve the diagnosis process in several ways. First, we can provide operators with an up-to-date overview of all components at the service, middleware and OS level that belong to a particular service composition. Second, information about the dependencies between these components can reveal which components are critically important for other components and for the overall service composition to provide correct service. Another potential benefit is that it helps to delineate the local infrastructure from any remote services used. This is likely to assist with the identification of issues whose cause lies in another administrative domain. Finally, a dependency graph can facilitate the selection of which components to investigate more closely. It can guide the diagnosis process along the chains of dependencies of a component that displays failure symptoms while enabling an operator to exclude other components that are independent from problematic ones. An explicit model of system structure can assist system operators in navigating the initially large problem space they are faced with.

Analysing software systems to determine dependencies is a known software engineering problem. Program dependence graphs (PDG) [Ferrante et al., 1987] represent data and control dependencies between the statements of a single procedure. Each node in a PDG represents a statement and edges represent either a dependence of the operation performed by this statement on a data value or a control condition over the execution of this statement. Similarly, system dependence graphs (SDG) [Horwitz et al., 1990] extend PDGs to the inter-procedural case. An SDG combines the PDGs of a multi-procedural program into a single graph whose edges represent calling and parameter dependencies between procedures or methods. Dependence graphs are generated based on techniques for control and data flow analysis, which are either static or dynamic. Static approaches operate on a representation of the program code and therefore require access to its source code. Dynamic analysis techniques rely on traces of program execution obtained at runtime, which usually necessitates the instrumentation of a program to profile its operation. The level of granularity of such dependence graphs is that of procedures and of statements within procedures. The resulting graphs can be used to improve human understanding of a program. However, for large programs that make extensive use of libraries and hence result in complex graphs they are primarily useful as a program representation that supports automated software engineering operations, such as various compiler optimisations.

Service compositions demand a different approach. First, control and data dependencies are at too fine a level of detail to be able to provide a useful overview of an entire service composition. There is a large number of components that would need to be analysed and each of these components consists of multiple libraries that together specify a large number of classes and methods. Instead, a model that captures the general system structure in the form of its major components seems to be a more appropriate level of detail for supporting an initial root cause localisation of failures in a service composition. An initial diagnosis based on our model of system structure can still be followed up by the use of more fine-grained system models, once a smaller set of root components has been identified. Second, we cannot assume access to source code beyond published service interfaces and possibly the backend implemen-

tations of services in the local administrative domain. Furthermore, it would be preferable to avoid the instrumentation required to obtain detailed execution profiles.

A typical middleware-based system is comprised of numerous components that can be deployed across various hosts. These components can be regarded as modules of functionality that are in part interdependent on each other. For example, a database management system relies on a file system and a web server needs the network interfaces of its host operating system to work. These individual building blocks are arranged as needed in order to support a particular application, which imposes a structure of relationships between them. In order for the components to be able to find each other and be able to use each others' features, information about their deployment has to be reified in some manner. In many middleware-based systems such information is made explicit in the form of various deployment descriptors and configuration files[1]. These describe some of the deployed components and their relationships to other components. Furthermore, there exist default containment relationships that are known to exist between certain types of components. For example, a BPEL process is deployed within some kind of BPEL runtime and an Enterprise Java Bean requires the services of the container it is deployed into. And thirdly, service compositions often specify their partner services as part of their specification in a composition or orchestration language.

This reification of deployment information allows us to automate the discovery of components that are relevant to a given service composition along with the dependencies they form on each other. In our approach, we exploit existing sources of static and dynamic information about the deployed components and their relationships. We furthermore perform a static analysis of any service composition artefacts. By relying on the processing of information sources that are readily available in many middleware-based systems and service compositions, we avoid the need to introduce intrusive instrumentation as might otherwise be required to, for example, trace the traversal of requests between components. We complement the results of these analyses with knowledge about default relationships that is encoded in per-component models. It is then possible to test a given deployment for the existence of the containment dependencies indicated in the available models.

4.2 Structural Cross-Domain Dependency Graphs

A structural cross-domain dependency graph represents the structure of a service composition from the viewpoint of an administrative domain as is shown in Figure 4.2. It consists of the major components that comprise a given service composition and their dependencies. In our model, the components of the operating system only form dependencies among each other on a single host (we do not, for example, capture dependencies on NTP or DNS servers), while middleware components can also have dependencies across hosts. At the service-level dependencies can span domain boundaries, but for components that are in other administrative domains, visibility is limited to the level of Web services.

Our model of system structure consists of the major components that comprise a service composition and their dependencies. We have already explained our understanding of components in Section 2.1.2, but repeat part of that discussion here as a reminder. We consider components to be services

[1] We will discuss exceptions to this in Section 4.6.

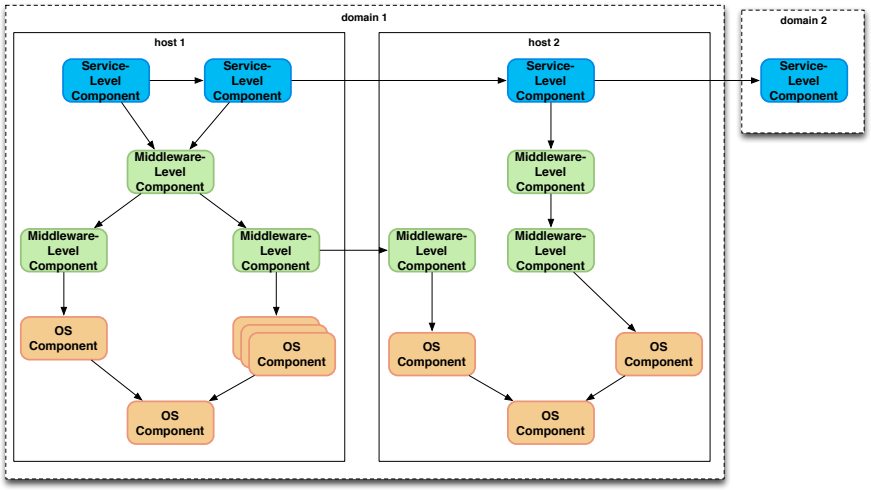


Figure 4.2: The components and dependencies represented by a structural cross-domain dependency graph.

at the service level, software modules at the middleware level and the resources and hardware components managed by the operating system. We also consider their major subsystems to be components. For example, components at the service-level are service interfaces and service composition artefacts, such as BPEL processes. At the middleware level we consider a database management system as well as the databases it manages as components. Similarly, an application server and its memory subsystem are related, but separate components. And finally, at the level of the operating system we have components such as network interfaces, file systems, memory and CPU.

The basic elements of our structural dependency graphs are components and dependencies. A component represents a particular type of component. The set of possible types is open-ended and examples of component types are a Web service, an operating system process or a SOAP runtime. Experts are responsible for defining component types in per-component models. These models describe a component type in terms of a set of relevant metrics, attributes and default containment relationships that commonly exist between this and other types of components (we discuss this in more detail in the next section). The set of attributes represent various properties that are specific to an instance of this type of component. Examples of attributes are the installation path of a middleware component or the port number at which it accepts requests. Each instance of a component type is assigned a key that uniquely identifies it within an administrative domain.

Structure is provided through the dependence relationships between components. Our dependency graph captures three types of dependencies that fall into two categories; *deployment* relationships among components at all levels and *calling* dependencies between Web services at the service level. Deployment dependencies have been established at the time of deployment and indicate that the first component needs some functionality offered by the second component in order to provide its service correctly. Our model does not capture more specific information about the type of relationship between two components, such as the function or role a component fulfils opposite other components. A deployment dependency can

be of any of two types: *containment* or *explicit*. A containment dependency represents a default containment relationship between two components and results from the manner in which components have been arranged in a given deployment (i.e. one component is deployed within another). For example, a default containment relationship can exist between a component of a web application and an application server that serves as its container. When a component makes a reference to another component that it relies on in some way, this gives rise to an explicit deployment dependency. An example of this is a Grid job scheduler that has been configured to make use of a particular compute cluster. Explicit dependencies allow our graph to capture dependencies other than simple containment relationships between components that would otherwise appear unrelated. At the service level, *calling* dependencies indicate that a service makes use of at least part of the interface of another service and thus represent potential invocation paths among partner services. All three types, calling, containment and explicit dependencies, represent a uni-directional relationship between exactly two components.

A dependency also has an extent. It can exist between two components on the same host, between components on two separate hosts and at the service level dependencies can span across administrative domains. We refer to these as intra-host and inter-host as well as cross-domain dependencies.

Our model of cross-domain service compositions is modular in order to address issues of scale and lack of control. In light of a potentially large number of distributed components that need to be discovered and processed, it may be necessary to divide a large dependency graph into smaller sub-graphs. For example, an overall graph can be divided into sub-graphs on a per-host basis. In our model we have a special type of component, which we refer to as a *placeholder* component. A placeholder component records just enough information to uniquely identify a component that is not part of the local sub-graph. If there is a dependency from a component in the local sub-graph on a component in another sub-graph (e.g., between components on two different hosts), then a placeholder component affords the immediate registration of this dependency, without, at the same time, having to discover the remote component and its potentially substantial sub-graph. For example, a placeholder can represent a dependency between a client on one host and a GridSAM server on another host. The remote GridSAM server can be identified by a combination of its host, component type and port number under which it accepts requests. The placeholder component provides sufficient information to resolve this dependency on the remote component in case the sub-graphs need to be merged for visualisation or if some other form of processing becomes necessary at a later point in time. For example, a placeholder component can represent a Web service on another host in the same domain and use the endpoint URL of that Web service to identify it.

Another issue that demands modularity is the lack of control over the dependency and discovery processes that may be in place at remote administrative domains or, in fact, whether any such information is maintained and made available. The *Inter-Domain Component* (IDC) serves as a local representation of a Web service in another administrative domain. It allows us to record a dependency from a local component onto a service in another administrative domain and identify the remote Web service via its endpoint URL. The dependencies between the credit card processor and the services of the two banks

and the reliable storage provider in Figure 4.1 would be represented as IDCs from the viewpoint of the domain of the credit card processor. Any relevant attributes of the remote service and any measurements that are available for it can be associated with the corresponding IDC and stored in the local domain. This enables us to record service-level dependencies across domain boundaries and to do so regardless of whether or not service providers maintain and offer such dependency graphs of their services. Furthermore, an IDC represents a dependency at the service level and hence does not reveal any sensitive implementation details. Together, the placeholder component and IDC cater for the modularity of our model and make the discovery and processing of large graphs more tractable.

Our dependency graphs capture the structure of a service composition in the form of a directed graph, $G = \{C, E\}$. The set of vertices, $C = \{c_0, c_1, \ldots, c_n\}$, is the set of instances of component types that have been discovered in a given deployment, each with an identifier that uniquely identifies this component within its administrative domain. There are three special types that can be assigned to any component. These are $ROOT, PLACEHOLDER$ and $INTERDOMAIN$. The first one, as its name suggests, denotes a root component. In our dependency graph there is one component per host on which all others in the system depend and this component is typically the operating system on that host. The other two represent placeholder and inter-domain components and are used by the discovery process to identify places where sub-graphs are to be merged. The set of edges is the subset of the Cartesian products of the set of components and their dependency relationships, $E \subseteq C \times D \times C$. $D = \{d_0, d_1, \ldots, d_m\}$ is a finite set of dependency types that can exist between two components. As discussed, the three types are $CALLING, CONTAINMENT$ and $EXPLICIT$.

The following properties hold. The binary relationship between two components given by any $d \in D$ is uni-directional and hence not symmetric. That is, $(c_i, d, c_j) \in E$ does not imply $(c_j, d, c_i) \in E$ for all $d \in D, c_i, c_j \in C$. Our dependency graph does not capture reflexive dependencies. That is, $(c_i, d, c_i) \notin E$. Dependencies can be transitive; if (c_i, d, c_j) and (c_j, d, c_k) then it follows that (c_i, d, c_k). Our dependency graph only captures one type of dependency from one component onto another; if $(c_i, d_a, c_j) \in E$ then $(c_i, d_b, c_j) \notin E$ for all $d_a \neq d_b$. Finally, the dependency graph can contain cycles. That is, $(c_i, d, c_j) \in E$ and $(c_j, d, c_i) \in E$ can hold.

We should briefly explain why we use the term "system structure" and "structural" to describe our model instead of simply referring to "system architecture", which is a term readers may be more familiar with. In the latest ISO systems and software engineering standard [ISO/IEC/(IEEE), 2011] architecture is defined as "*fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution*". The FAQ accompanying this standard [iso, 2011] furthermore explains that the overall structure or organisation of hardware and software that make up a system is not considered to be an architecture. As this kind of organisation is what we are essentially interested to discover about a system, we largely refrain from using the term architecture to describe our dependency model. However, as we will discuss in related work, there is a body of work on so-called architecture recovery that shares certain commonalities with our efforts.

4.3 Deployment Information Base

Middleware-based service compositions provide numerous sources of information that we can exploit for the discovery of components and their dependencies. Some sources provide information about both the presence of components and their dependencies, while others only reveal one of these aspects.

The *specification of a service composition* is usually expressed in a suitable language. Such composition languages represent a useful source of information about service-level dependencies. Two primary paradigms for the specification of Web service compositions are orchestration and choreography [Peltz, 2003]. With orchestration control is centralised and the control and data flow among a set of partner services is defined in the form of a workflow. Choreography is its decentralised counterpart, in that interaction is specified in terms of message exchanges that can occur between pairs of Web services without involving a central focus of control. Both of these paradigms have in common that some mechanism is needed in order to identify partner services. For example, BPEL is an orchestration language. It contains constructs through which a BPEL process can identify any of the Web services it interacts with and also specifies which of their operations are going to be invoked. Similarly, WSCI [Arkin et al., 2002] allows the definition of Web service choreographies. Its language constructs allow a Web service to describe its interaction with other Web services involved in a choreography in terms of sequences of invocations on each others' operations. As long as services are statically bound to each other at deployment-time, a static analysis of a composition's specification can reveal calling dependencies between Web services.

Before programs can be executed, they are usually installed in some way. Most modern operating systems offer a *software package manager* through which users can install a variety of software packages from online repositories. These package managers maintain information about the installed programs, including the dependencies that they form on other software packages. As has been proposed in [Kar et al., 2000], this information can be used to discover some components along with their dependencies, even when they are not currently executing. For cases in which software package managers are not used, we can make use of the fact that some components deploy their libraries, executables and scripts in one of several default directories. By determining whether the given files for a component exist in *common installation paths*, we can detect components whose installation has occurred outside the mechanisms of a software package manager.

Many of the components that comprise the software stack of middleware-based systems rely on deployment mechanisms outside those provided by OS software package managers. One common mechanism used by many types of middleware are *deployment descriptors*. Deployment descriptors represent another useful source of information about components and dependencies. Conceptually, the contents of a deployment descriptor consist of several parts. The first is a set of files that constitute the functionality of a component in the form of libraries and executables. A so-called manifest provides meta-data about the component. This can include a component name and version, the URL under which it accepts requests and can also list other services and components it depends on for its operation. The manifest is used by the container into which the component is deployed in order to load it at runtime and make re-

quired services and other components available to it. Additionally, a deployment descriptor may contain additional configuration files and other resources required by the component.

There are several different types of deployment descriptors. Deployment descriptors in .NET are referred to as *assemblies*. An assembly consists of compiled code in the form of an executable file or a library and a manifest file. The manifest is written as a text file and once it has been stored within an executable, it can be accessed again with the use of utilities that are part of the Windows SDK. A .NET manifest provides name and version information, lists all files that the assembly is comprised of and, most importantly for our purposes, also specifies all other assemblies on which this one depends. A .NET assembly is deployed into a suitable runtime, which then uses the information in the manifest to determine which other assemblies it needs to load. Similarly, the Open Services Gateway initiative (OSGi) [Open Services Gateway initiative Alliance, 2012] allows for complex Java applications to be broken up into modules called *bundles*. Amongst other things, the manifest in an OSGi bundle specifies which packages it makes available to other bundles and which packages of other bundles it actually depends on for its functionality. Dependencies onto Java packages are at a different level of granularity than what we consider useful in our work, but it demonstrates the kind of dependency information that is available in middleware-based systems. There also exist simpler types of deployment descriptors. The deployment descriptor for an Enterprise Java Bean (EJB) identifies its interfaces and configures the services it requires from the container. The deployment descriptor for a Java servlet typically contains its compiled class files along with a simple manifest describing the application and some of its properties. For a Web service it stores a WSDL document defining its interface along with pointers to the backend implementation of that service. In such cases, the deployment descriptor may not reveal information about additional dependencies, but merely confirms the existence of a containment relationship between a component and the container into which it has been deployed.

The presence of a deployment descriptor confirms that a component of a particular type exists and establishes a containment relationship between it and the container. Furthermore, some deployment descriptors list dependencies onto other components in their manifests. Often these manifests are expressed in XML or they can be accessed through suitable utilities. Finally, by analysing configuration files that are part of the descriptor, it may be possible to identify specific subsystems of the container that the deployed component depends on (e.g., security or transactional subsystem).

There are readily available sources of dynamic data about executing components. The presence of some components can be confirmed by examining *operating system process tables*. A process is the OS representation of an executing program. It consists of a stack of instructions, data and a program counter indicating the current position in the execution of these instructions. An OS process table maintains information about all processes that are currently executing or that are waiting to be scheduled for execution. Among the information maintained in the process table are the names of processes. The name of a process usually identifies the program of which a process is an instance. By matching a string pattern that is unique to this type of component against the names of processes in the process table, we can identify its presence. The information in process tables allows us to detect components that are executing at

the time of discovery, but fails to detect others that are not currently active.

Two other sources of information are *configuration* files and *administrative interfaces*. Middleware components often allow for the configuration of several aspects of their operation through configuration files that exist outside of deployment descriptors. One of these aspects is to identify any other component that the given component relies on. This information can be used to discover dependencies that are not simple containment relationships. A small number of components offer administrative interfaces in the form of utility programs or APIs. It is often possible to query these interfaces for components that have been deployed within the component offering the administrative interface. Again, this enables the detection of components and confirms a possible containment relationship between them.

Finally, in addition to the information made available by middleware, the definition of a component type can specify zero or more default containment relationships. These specify which other types of components can typically provide a runtime environment for the given type of component and that it is usually deployed within. For example, a Web service runs within a SOAP runtime, which is itself usually deployed within some kind of application server. A human expert can encode per-component knowledge about such common relationships at the time of defining the component types. This makes it possible to generate containment hierarchies among the discovered components by determining whether the specified containment relationships do exist in a given deployment.

These sources of deployment information allow us to obtain an overview of the general deployment structure of a large service composition at the level of its major functional components and sub-components

4.4 Discovery Process

In this section we provide an overview of the process of discovering the relevant components of a service composition and their dependencies. The discovery process is a decentralised one, which enables it to adequately address the scale and decentralised nature of service compositions. We divide the task of discovering all components and dependencies within an administrative domain into smaller units of work by performing the discovery process on a per-host basis. Each local process is only responsible for the processing of components on its host and the discovery of its local sub-graph. As a result, it has to deal with a smaller number of components than would be the case for a domain-wide process. Furthermore, several processes can perform their discovery in parallel with each other. While the discovery process is decentralised, we do not address how to achieve decentralised storage and retrieval of the discovered data. This is an interesting issue for large deployments, but we consider it outside the scope of this thesis. Instead, each local process submits its sub-graph to a domain-central process, which is responsible for combining the local results into a domain-wide dependency graph. The resulting graph represents the structure of a service composition from the viewpoint of a particular administrative domain. It consists of the components and dependencies that exist at any level within this domain, but its visibility of components in other administrative domains is limited to the service level.

The individual procedures of our dependency discovery process are listed in Figure 4.3 along with the types of input data they operate on and their results. Figure 4.4 illustrates how each procedure adds

additional detail until we obtain a structural cross-domain dependency graph.

The first procedure is called *Component discovery*. Its input consists of the list of hosts in the administrative domain that are to be examined. At this stage the local discovery process on each host is driven by the per-component models that define individual types of components. One part of these component models are component-specific discovery actions, which encapsulate knowledge on how to exploit some of the sources of deployment information in order to discover instances of this component type. The local discovery process iterates over all component models and executes the corresponding discovery actions. Once it determines that an instance of a component type does exist on the given host, it assigns an identifier to this instance that is unique within its administrative domain. The discovery process proceeds to retrieve and store further relevant attributes about this instance. A representation of the discovered component is then added to the inventory, which will form the sub-graph of components on this host.

The result of the component discovery procedure is a list of all components that are deployed on each host in the administrative domain as shown in Figure 4.4(a). This involves components at all layers, but does not yet include any of their dependencies and as such does not convey any structure. At the end of this phase, we have the set $C = \{c_0, c_1, \ldots, c_n\}$ of all components that are deployed across the hosts within this administrative domain. The procedure has also assigned one root component per host in this domain; $c_i = ROOT$ for some $c_i \in C$.

The next procedure, *Processing of system-level dependencies*, uses the list of discovered components as input in order to process any dependencies that may exist among components beneath the service level. For each discovered component the corresponding model is consulted for any potential containment relationships that need to be evaluated. For example, the component type model of a Web service indicates that instances of this component type are usually hosted within a SOAP runtime. Based on this information the local discovery process can then check whether any discovered SOAP runtime actually contains this particular Web service. If a potential containment relationship exists in the current deployment, a corresponding dependency is recorded between the two components in the local inventory. In addition to this, a component model may specify component-specific procedures that can be used to determine dependencies that are not simple containment relationships. An example of such a component-specific procedure is to parse a specific set of configuration files. In case a dependency onto a component on another host is detected, this is recorded by inserting a suitable placeholder component into the inventory. No further discovery along this branch is necessary as the graph of the remote component is the responsibility of the discovery process on that host.

At the end of this phase, the per-host graphs represent containment relationships and other types of dependencies among components beneath the service level as is shown in Figure 4.4(b). Dependencies may span across hosts, but are confined to within a single administrative domain. The resulting graphs include dependencies from the discovered service-level artefacts onto the middleware components that support them. As such, the graphs resolve dependencies from the service-level down to the operating system. This procedure is likely to have assigned some components to be placehold-

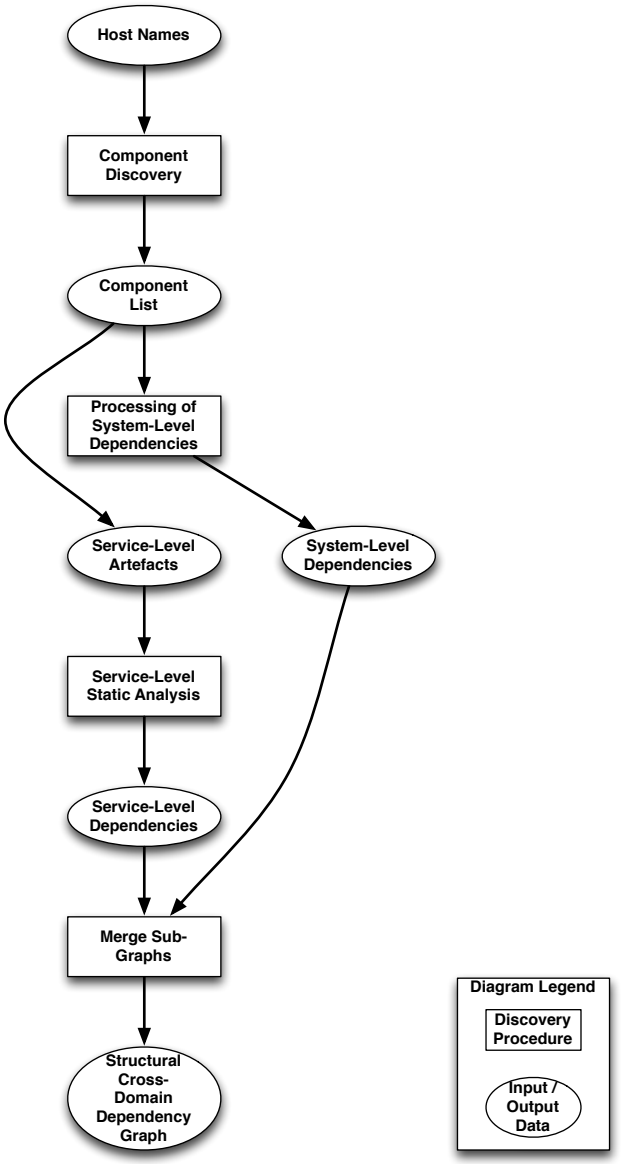
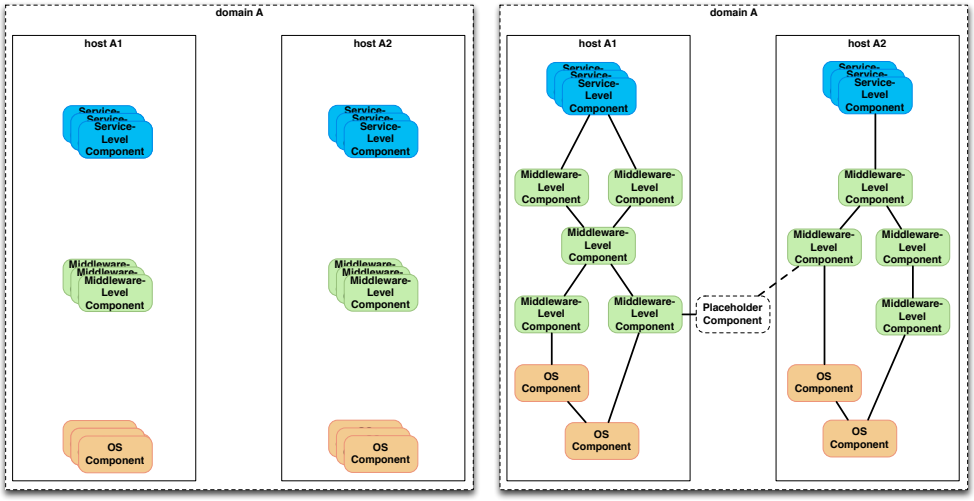


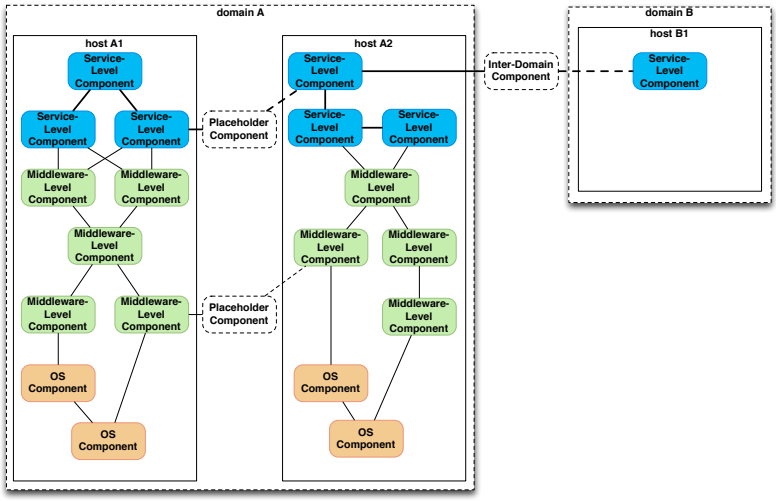
Figure 4.3: Overview of the procedures of the discovery process and their input and output data.

ers, $c_i = PLACEHOLDER$ for some $c_i \in C$. Furthermore, the edges in $E \subseteq C \times D \times C$ are based on dependencies of type $CONTAINMENT$ and $EXPLICIT$. At this stage the graphs still omit relationships among service-level artefacts.

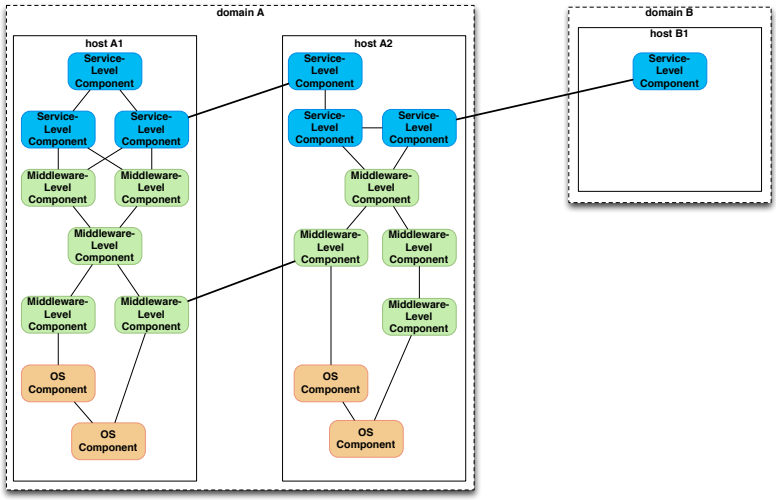
Static analysis is performed on any of the discovered service-level artefacts, which are a subset of the discovered list of components from each host. Each type of service-level artefact, such as for example orchestrations represented by BPEL processes and individual Web services expressed as WSDL descriptions, require custom analysis procedures. However, they all share the same basic technique. The service-level artefact, which usually exists in a human-readable format, is parsed in order to identify its partner services. The specifications of local partner services are analysed recursively, while dependencies onto services on other hosts and in other administrative domains are represented by a corresponding placeholder or inter-domain component.



(a) Component list (b) System-level Dependencies



(c) Service-level dependencies



(d) Structural cross-domain dependency graph

Figure 4.4: Discovery of a structural cross-domain dependency graph.

We present the details of our static analysis procedure for BPEL service compositions in Appendix A.3, but present a brief summary at this point. A BPEL process defines the set of partner services it interacts with in a set of so-called partnerLinks. Each of these partnerLinks is identified through a unique namespace that is matched by an import element in the BPEL process. An import element points to the location of the document defining the corresponding namespace. This makes it possible to locate and parse the WSDL documents that define the interfaces of the partner services used by a BPEL process. These documents define matching partner link type (PLT) definitions, which contain information that relates this partner interface to a series of other WSDL elements, such as the definition and suitable operation signatures and eventually to a service element that binds the partner service to a particular endpoint address. This allows us to resolve a dependency from a BPEL process onto an actual instance of a partner service at a particular URL. Following is an overview of the steps of this static anlysis procedure. Further details and examples are given in Appendix A.3.

1. A given BPEL process, B_1, is parsed and a mapping of prefixes to their corresponding namespaces is created based on the information contained in the preamble of B_1.

2. A second mapping is created. This time from the namespaces to the location of the documents that define these namespaces. This mapping is based on the information contained in the import elements of B_1.

3. Next, the parterLink definitions in B_1 are analysed. For each partnerLink, its name, reference to a PLT definition and PLT namespace prefix are recorded. The namespace prefix is resolved to its actual namespace and from that to the location of its corresponding WSDL document. A third mapping is generated. This time from the names of the partnerLinks to the locations of their WSDL documents.

4. For each partnerLink, the corresponding location is accessed to parse its WSDL document, WS_2. WS_2 either defines the partner link type of this partnerLink directly or imports another document that defines it.

5. Once the corresponding PLT definition has been found, it is analysed to determine the portType it references. This information is used to look for a binding element with a matching type in WS_2 or in one of its imported documents. The name of the binding is in turn used to look for a matching service element. If such a service element is found, a dependency onto the service at the endpoint URL specified in the service element is recorded.

6. The domain-central part of the process performs an additional step during the merging of the local sub-graphs. If the target namespace of WS_2 matches the target namespace of another BPEL process that has been discovered, then this represents a dependency between two BPEL processes, namely of B_1 onto BPEL process B_2 whose Web service interface is represented by WS_2. Otherwise, the dependency is between the BPEL process B_1 and the Web service WS_2.

This process is repeated for each partnerLink and for each BPEL process.

This static analysis further refines the local graphs through the inclusion of dependencies at the service-level. Figure 4.4(c) shows additional dependencies among the service-level components. The set of edges, E, has been extended with dependencies of type $CALLING$. The procedure may also have taken account of any dependencies on services in other administrative domains by inserting inter-domain components, $c_i = INTERDOMAIN$ for some $c_i \in C$. At this stage we have a set of structural dependency graphs, one for each host in the domain, along with placeholder and inter-domain components as the links between these sub-graphs.

The final step consists of the *merging* of these sub-graphs. The local sub-graphs are submitted to a domain-central process, which resolves any placeholder components and replaces them with the representations of the components they point to. The inter-domain components are left in place as they will serve as the local representation of their remote Web services. The end result is a structural cross-domain dependency graph. As shown in Figure 4.4(d) this graph uncovers all the components that are involved in a given service composition. It shows dependencies among services even across domain boundaries and the dependencies on and among middleware components down to the operating system.

4.5 Cross-Domain Measurements

Our cross-domain dependency graphs have additional uses beyond serving as an aid to understanding the structure of a service composition. One other issue it enables us to address is how to exchange measurements among service providers and their clients. As we have already discussed, an overloaded or unavailable service can impact other applications and services that depend on it either directly or via chains of dependencies along which a failure propagates. Therefore, system operators need to be able to routinely evaluate certain characteristics about the operation of the services their applications depend on. It is desirable to enable communication of certain measurements between service providers and clients in order to facilitate this. In fact, several providers of Internet services have begun to provide overviews of the status of their services and infrastructure in the form of online dashboards (e.g., http://status.aws.amazon.com/) that offer data for human consumption.

Such an approach is not sufficient for service compositions due to a number of reasons. First, the potential scale of service compositions in terms of the number of clients and service providers demands an automated solution to the exchange of measurements. System operators should have access to detailed information about the performance of relevant Web services without having to manually search for information on various web sites. Service providers need to disseminate these data to their clients on a continuous basis. Second, given that service compositions are generally loosely-coupled, it is not practical for service providers to maintain up-to-date lists of all the clients that need to be informed of new measurements. Third, as trust does not usually extend beyond administrative domain boundaries, service providers must be enabled to maintain control over what information is published so as to prevent the release of sensitive information about the implementation of their service offering. Finally, we want to avoid clients having to continuously poll for information about services in remote administrative domains as this is inefficient and can result in heavy loads on service providers.

Cross-domain dependency graphs enable us to overcome these issues. Given the information con-

tained in these graphs, clients are able to identify their cross-domain dependencies and as such the services whose metrics they need to monitor. Service providers, for their part, can offer published interfaces through which clients can subscribe for notifications of updates to measurements. This relieves service providers from having to maintain lists of clients to communicate measurements to. Instead, clients are responsible to subscribe for updates to their dependencies. Furthermore, providers are free to choose which clients get access to updates about particular metrics and maintain control over what information they expose. The exchange of measurements can be controlled by the use of general as well as client-specific policies. Clients receive up-to-date information about the services they depend on and no unnecessary traffic is generated as clients rely on a subscription mechanism to get notified when new data is available. We have previously proposed a solution for the implementation of cross-domain change management processes based on similar principles [Wassermann et al., 2009].

4.6 Limitations

Our dependency discovery approach is not suitable for all types of middleware-based service compositions. One widely-used model in service-oriented architectures, for which our approach fails to discover service-level dependencies, is that of an Enterprise Service Bus (ESB). An ESB is a type of message-oriented middleware that caters for the integration of heterogeneous applications and services. An ESB provides numerous services, but of primary concern for our purposes are its messaging capabilities. Endpoints use the bus to communicate with each other in an asynchronous manner via message queues. In addition, an ESB adds the capabilities of a message broker thereby removing the need for point-to-point connections between endpoints and relieving senders from explicitly identifying the intended recipients of a message. Instead, the message routing logic can be placed within the bus to be applied to all messages that match certain conditions. These conditions can be in the form of rules about the message content. The bus can then deliver messages into the correct queues where they are consumed by corresponding endpoints.

While discovery of the services that are installed on the bus may be possible by detecting so-called adapters that transform messages on behalf of an application or service into a format it understands, our approach cannot determine the relationships between services given this level of decoupling. It would be necessary to instrument the message queues in order to determine the identities of the producers and consumers of messages and thereby attempt to infer the dependencies among them.

Our approach will also fail to discover service-level dependencies of applications that employ RESTful interfaces. As these interfaces identify only data resources, there is no need to specify partner services as such. Unless there is some representation of a sequence of resource accesses that are to be made as part of a composition of RESTful services, as may be possible using services such as Yahoo Pipes [Yahoo!, 2012], there is no way to determine service-level dependencies in a static manner without tracing the exchange of HTTP requests at runtime.

We will review the accuracy and completeness of the resulting dependency graphs in the next chapter, but want to note two further limitations at this point. In its current form, our approach does neither detect data dependencies that components may form on each other or on elements of the file system nor

does it discover calling relationships beneath the service level among the components comprising the middleware and operating system.

4.7 Related Work

Existing approaches to dependency discovery in software systems can be divided into indirect and direct ones. Indirect approaches discover dependencies dynamically through observation of some aspects of the runtime behaviour of a system. The resulting dependency graphs are functional ones that capture the relationships between entities as they have been exercised during execution. In our review, we include some work on architecture recovery in this category. Direct approaches, such as ours is one, are based on models and exploit available sources of information to discover structural models of a software system.

4.7.1 Indirect Dependency Discovery

Indirect approaches to dependency discovery typically apply probabilistic models to runtime observations in order to derive likely dependencies. In [Gupta et al., 2003] the authors present such an approach to discover likely dependencies between the components of a web application. The components considered by the approach are URLs, servlets, EJBs and database queries. It exploits performance metrics as they are commonly available in middleware-based systems and avoids the need to retrofit systems with intrusive instrumentation. The approach needs to be able to record invocations and their execution times in order to measure the request processing times expended by components. Dependencies are determined by identifying containment of activity periods. If a component A invokes another component B and A finishes its execution only after the invocation of B has returned, then this indicates that B's activity period is contained by A's and points to a possible dependency. In order to determine the probability that this containment is representative of a dependency, the approach calculates the proportion of requests to component A that also result in a contained invocation to component B. The approach has maximum confidence in the presence of a dependency, if all requests to A result in invocations to component B. The drawback of relying on activity period containment is that as the degree of concurrency and therefore the interleaving of request processing increases, so does the rate of false dependencies.

A similar approach that attempts to reduce the rate of false dependencies is presented in [Kashima et al., 2005]. Messages that are exchanged between components are recorded at network devices (e.g., routers and switches). Matching pairs of intercepted HTTP requests provide the start and end times of transactions. The Expectation-Maximization algorithm [Dempster et al., 1977] is used to compute maximum likelihood estimates on the set of components and the observed transaction times. The resulting estimates are used to infer direct dependencies between two components. The approach also takes into account the number of times that one component calls another one. In order to reduce the rate of false dependencies, the authors propose to estimate the probability with which the messages between two components are only contained by chance based on the arrival times of requests and the processing times at services.

The Orion dependency discovery system [Chen et al., 2008] discovers dependencies between hosts and common network services, such as the Domain Name Service and Kerberos. Orion analyses IP, TCP

and UDP headers to obtain timing information about network traffic between hosts and these services. It makes use of an observation about the distribution of network delays between network services. The distributions between dependent services are not random, but instead exhibit spikes that are correlated with the processing delays introduced by these services and with the network delays between them. Based on this observation, Orion calculates the delay distributions for sets of network messages as they occur within a time limit that is larger than the response time of any of the services. The resulting dependencies are based on correlation and may therefore indicate dependencies that do not actually exist. In order to model distributions with statistical validity, Orion may need to collect several days' worth of network traffic.

Active Dependency Discovery (ADD) [Brown et al., 2001] can dynamically discover dependencies among components deployed in an application server. ADD assumes that a repository of information about existing components exists in some form of configuration management database and that the components can be monitored for their response times and availability. The idea is to systematically perturb individual components through fault injection and observe which other components exhibit statistically significant deviations in their performance. If such a deviation is detected, then it is assumed that there is a dependency between the perturbed component and the one that has slowed down or has become unavailable. The perturbation applied in the evaluation of ADD is to periodically lock various database tables, while workload is applied to the web application under examination.

Automatic Failure-Path Inference (AFPI) [Candea et al., 2003] represents an improvement over this approach. The goal is to discover how failures propagate among the components of a web application. All EJBs within an application server are discovered and the Java Reflection API is used to discover all of their methods and declared exceptions. Some additional input/output and memory-related exceptions are added to those that have been discovered. The application server is instrumented to allow for the injection of exceptions upon the invocation of each method. This enables AFPI to systematically inject the discovered injections into all methods of the EJBs. Simple failure detectors record the corresponding failure of all other components and a so-called failure map is build up iteratively in this fashion. An edge in the failure map between two components indicates that an injected failure in one component has propagated to the other component by causing it to fail as well. This relieves a developer from having to reason about how to perturb the system. The authors find that AFPI can detect more dependencies along which faults can propagate than they could do manually by analysing deployment descriptors.

In [Basu et al., 2008] the authors present three techniques to dynamically discover dependencies between Web services by analysing data about execution times and the exchanged messages. A SOAP runtime is instrumented to record the messages being exchanged between services. Three simple techniques are discussed. The first one calculates conditional probabilities for the occurrence of a dependent message. If, after a service has sent one message, another service sends a second message within the execution time of the first one and if this happens sufficiently often, then a dependency is assumed to exist (i.e. high conditional probability). The second technique analyses the distribution of service execution times. Assuming that that the differences in times between incoming and outgoing messages follow a

normal distribution, it is possible to test for a fit with the normal distribution and record a dependency, if the test value exceeds a certain threshold. The third technique creates histograms of the time differences of all pairs of messages. The authors have found that a small number of consecutive buckets with counts that considerably exceed the average count is a good indicator of a dependency. The end result of each of the three techniques is a dependency graph whose edges provide the probability with which it represents a real dependency. This allows the graph to be further pruned by removing edges with low probabilities.

DiscoTect [Schmerl et al., 2006] is an approach to software architecture recovery whose goal is to enable verification that an implementation adheres to its designed architecture. DiscoTect represents another indirect approach as it is based on an analysis of the runtime behaviour of a software system. The monitored runtime events can be things such as method calls, CPU utilisation, network bandwidth and memory consumption and so on. In order to bridge the gap between these low-level monitored events and architectural information, the authors have developed the DiscoSTEP language, which allows users to specify mappings between monitored and architectural events. Rules define how basic events can be composed into more complex sequences that represent architectural information. For example, the initialisation of an object can be regarded as a basic event and a DiscoSTEP rule can determine that it represents the creation of a Server component. A subsequent event that denotes the creation of a socket by another object to the object representing this Server component can then be converted into a relationship between a Client component and the Server component it communicates with. Such mappings might be reusable due to their generic nature. However, the success of this approach depends on programmers following certain conventions in their implementation.

In [Maqbool and Babri, 2007] the authors examine approaches that apply hierarchical clustering to software architecture recovery. Hierarchical clustering is a bottom-up approach that determines the similarity of different entities by calculating distances between their features. Similar pairs of entities are then merged into progressively larger clusters. Clusters reveal functions or classes that share similar features and are therefore likely to be related to each other. For example, the analysis of a simple HTTP server written in C or Java might result in clusters that correspond to its socket subsystem, its TCP/IP module or a control component that maintains a set of threads.

Indirect approaches afford the discovery of dynamic dependencies. They produce functional graphs, which represent relationships that are indeed exercised. Due to the probabilistic manner in which they are generated, they may contain false dependencies. They are also only able to discover dependencies that are exercised under a given workload. However, this does not distract from their utility in helping to understand the relationships in a complex system. Our approach is a direct one in that it exploits deployment information as it is available in most middleware-based service compositions without necessitating intrusive instrumentation. The resulting structural dependency graphs show only dependencies that exist in a given deployment (no false dependencies), but may omit to include relationships that are not part of the available model knowledge or that cannot be discovered from the available sources of deployment information. Work on software architecture aims to support developers with understanding how the architecture of a software system has evolved since the initial creation of its design documentation. The

level of granularity that is common to these approaches (i.e. functions or methods and variables) is too detailed for our purposes.

4.7.2 Direct Dependency Discovery

There are far fewer direct approaches to the discovery of dependencies in software systems. In [Kar et al., 2000] the authors propose a direct approach to automated dependency discovery by exploiting information that is available from software package managers in operating systems. These usually contain information about installed software along with information about their dependencies on other applications and libraries. Some software package managers (e.g., AIX Object Data Manager) contain information that allows to infer dependencies among components on different nodes. The authors describe a simple architecture to discover dependency information in this manner and to store it in a database for use by application service agents.

Another direct approach is presented in [Magoutis et al., 2008]. Galapagos is a system for the discovery of application data relationships. Users encode model knowledge in the form of so-called Data Location Templates (DLT). There is one DLT per software component. Each DLT specifies the locations of data items and how the component makes this data available. A DLT can refer to an executable script that encapsulates procedures for the extraction of data-specific information from its associated software component. These scripts can make use of a variety of mechanisms to determine, if a modelled data dependency does in fact exist (e.g., use API of management subsystem or of application server). A distributed crawling algorithm can then process these models and make use of the scripts to build a map of all application-data associations in a given deployment. The algorithm assumes that deployed components have already been discovered by existing tools and that it can access information about them from a configuration management database.

Our approach builds on the one by [Kar et al., 2000]. We extend the proposed approach by considering and integrating additional sources of information about deployed software components and their dependencies as they exist in many middleware-based service compositions. Furthermore, our approach exploits service composition artefacts in order to discover cross-domain dependencies at the service-level. The approach used by Galapagos is similar to the one we have adopted. We use per-component models that point to executable discovery routines. Galapagos might represent a suitable approach to augment our graphs with information about the data dependencies that some components form. We do not have sufficient information about the implementation and operation of commercial tools to review them fairly.

Conclusions

In this chapter we have discussed a way in which to make the inherent structure of a service composition explicit and represent it in the form of structural cross-domain dependency graphs. We have given a characterisation of these as directed graphs that capture the major components comprising a service composition along with the dependencies among them. Dependencies at the service level represent calling relationships among Web services and can extend beyond administrative domain boundaries.

Beneath the service level, our dependency graphs also capture dependencies as they have been formed at the time of deployment. After listing the different sources of deployment information that are available in middleware-based service compositions we have given a procedural overview of how we can exploit this information in order to automatically generate dependency graphs for a given service composition. We have explained how the process begins by discovering deployed components based on models that encapsulate knowledge about component types and how additional detail is added through subsequent stages of the process, which also involves static analysis of service-level artefacts in order to discover the dependencies between services. We have briefly discussed why it is important to enable the exchange of certain metrics among service providers and their clients and why it is difficult to do so efficiently across domain boundaries. The structural dependency graphs our method obtains can overcome the identified issues. Finally, we have compared our efforts with related approaches to direct and indirect approaches and with some work on software architecture recovery.

So far we have described our approach to monitoring a service composition and to making information about its structure available. We have motivated this work with arguments as to how these approaches may improve the process of detection and diagnosis of system-level failures. Before we consider detection more closely in Chapters 6 and 7, we first evaluate our proposed approach to data-driven diagnosis. We want to establish the feasibility of the mechanisms we have outlined and determine their cost. We can then find out under what circumstances data-driven diagnosis can provide effective support to system operators when dealing with system-level failures.

Chapter 5

Data-Driven Diagnosis

In this chapter we evaluate the feasibility and the effects of data-driven diagnosis. We have developed a prototype that enables this approach through the comprehensive monitoring of service compositions and the discovery of components and their dependencies as described in the previous two chapters. We have deployed a real-world cross-domain service composition on a distributed testbed, which we use for our experiments. In order to establish the feasibility of our approach, we carry out a performance analysis of the overheads it introduces and then discuss the accuracy, completeness and limitations of our dependency graphs. Next, we quantify the benefits of data-driven diagnosis in terms of the success rates and diagnosis times it enables system operators to achieve. For this purpose, we have performed a controlled experiment that involves human participants who serve as system operators. We compare the performance of participants in an effect group using data-driven diagnosis to that of participants who rely on a traditional approach using a standard tool set. From the results of this experiment, we learn what failures data-driven diagnosis is most effective for and under what circumstances its application is justified and benefical. We conclude this chapter with a discussion of the insights we have gained from this evidence-based determination of the effects of such an approach to failure diagnosis.

The key results we are able to report on in this chapter are as follows.

- We carry out a performance analysis of our monitoring prototype, which reveals moderate to small overheads. The average slowdown to a monitored application is below 10%. The resource requirements of monitoring agents are modest with a CPU usage of about 3% to 4% and memory usage of about 20 MB. Finally, the communication overheads arising from both intra- and cross-domain monitoring network traffic are small enough to be tolerated by modern networks.

- We confirm that data-driven diagnosis leads to a statistically significant increase in success rates. Participants who use data-driven diagnosis are able to achieve a near-perfect success rate compared to a success rate of between $\frac{2}{3}$ and $\frac{4}{5}$ for participants using a standard tool set.

- We confirm a statistically significant reduction in the diagnosis times achieved by participants using data-driven diagnosis. A reduction of about 22% is observed across all injected failures and this increases to savings of about 55% for failures whose cause lies in another administrative domain.

- An inspection of diagnosis times for individual failures reveals time savings for most of them based on data-driven diagnosis.
- We are able to make observations about the utility of structural cross-domain dependency graphs to guide the diagnosis process and gain insight into the negative effects of its limitations (i.e. missing relationships between components) on the performance of system operators.
- Based on our results, we reason about the circumstances under which the application of data-driven diagnosis is most beneficial.

5.1 Monere Prototype

We provide a brief overview of our prototype framework, *Monere*, that implements the features described in the previous two chapters. Please refer to Appendix A for additional implementation details. Appendix B.3 reproduces the instruction materials that participants using the data-driven approach were provided with. It presents an overview of how to use our prototype to perform failure diagnosis and contains numerous screenshots of its user interface.

The Monere prototype is based on an early version of the open-source RHQ enterprise management system [Red Hat, Inc., 2012]. RHQ provides a set of core services for systems management, such as an abstract inventory model and the discovery of some hardware and software components. Monere modifies and extends RHQ in order to make it suitable for cross-domain service compositions and to implement the features that we have discussed in the previous two chapters.

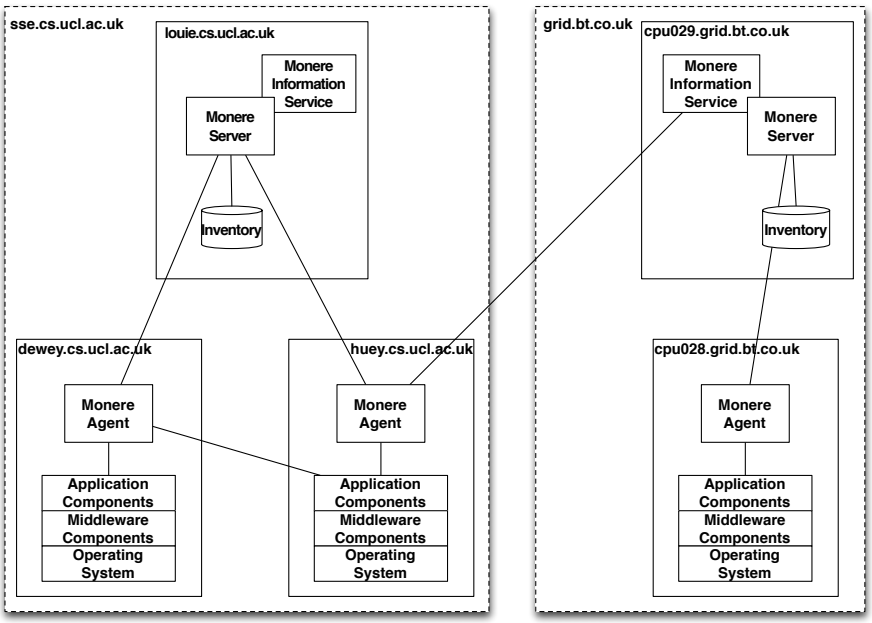


Figure 5.1: The key components of the Monere prototype as it is deployed on our cross-domain testbed.

The three main components of Monere are the Monere server, monitoring agents and the Monere Information Service (MIS). They are shown in Figure 5.1 as deployed across two administrative domains

in our testbed. Each domain hosts a central Monere server, which communicates with the set of Monere agents deployed on the hosts in its domain. The server obtains component discovery results and measurements from these agents and maintains an inventory for all hosts in its domain. The server persists the inventory in a database and makes this data available to end users via a web-based user interface. The Monere server is also responsible for processing and merging the individual sub-graphs discovered by its agents.

On each host there is a single Monere agent, which runs within its own Java process. It is responsible for the discovery of any components on its host and for the subsequent measurement of their metrics. An agent periodically executes the discovery process in order to detect any changes in the local deployment. Measurement collection is driven by user-defined collection intervals for each metric. The smallest supported interval at which an agent takes measurements is one second. Suitable collection intervals for our case study have been determined experimentally and range from five seconds to ten minutes. An agent stores all measurements for the components it has discovered in a measurement report and then checks this report every 30 seconds to determine whether it contains new data that needs to be sent to the server.

An agent is essentially a runtime system for a set of plugins and manages communication with the Monere server. Once an agent has been deployed on a host, it registers with the Monere server in its domain and downloads any needed functionality in the form of plugins. A plugin represents a particular component type and encapsulates the functionality for its discovery and the measurement of corresponding metrics. Plugins consist of a descriptive part in XML and a Java implementation. The former specifies information about the represented component type and defines a set of metrics to be collected from instances of this component type along with data type definitions and user-configurable collection intervals. The implementation of resource discovery and measurement collection is performed by the corresponding class files referenced in the plugin descriptor.

The MIS implements our approach to the exchange of measurements between service providers and their clients across domain boundaries (cf. Section 4.5). It addresses the requirement of providing visibility of dependencies on components in remote administrative domains and sharing information about their state. A service provider can make metrics about its published services available to clients by publishing them as RSS feeds at the MIS responsible for its domain. An agent that has discovered a dependency from one or more of its local components onto a Web service in another administrative domain, can subscribe to the corresponding RSS feed at the MIS of that domain. The MIS restricts visibility to the level of published Web services and provides no further insight about the underlying infrastructure. It safeguards the sensitivity of implementation details, while providing data that can help to determine problems with remote services.

We identified the need to correlate application activity with measurements taken at the other layers. Monere implements a simple correlation mechanism based on timestamps. For each application activity Monere records its start and end time. Similarly, agents record timestamps for any other measurements they take. Clock synchronisation among agents and servers is performed using the Network Time Pro-

tocol (NTP) [Mills et al., 2010], which achieves synchronisation of clocks across the Internet to within a few milliseconds of each other. This degree of accuracy is sufficient as the smallest collection interval supported by Monere is one second. In order to correlate application activity with lower-level measurements, Monere calculates a time range of a few seconds on either side of the occurrence of an activity and then includes all measurements that fall within this time range as being correlated with it. While correlation based on timestamps cannot guarantee to establish causal relationships, it can afford insight into how different parts of the system reacted at the time the application performed a particular activity.

Monere's web-based user interface provides access to information about all components that have been discovered in an administrative domain as well as their service-level dependencies across domain boundaries. The UI is composed of a number of panels, each of which provides a different view onto a specific part of the data. The first three panels, shown in Figure 5.2 allow a system operator to navigate the discovered components. The Resource Tree view presents the discovered components in an hierarchical manner ordered by their hosts. System operators can use this to obtain an overview of all components and to navigate their containment relationships. The Dependency view displays the dependents and dependencies of any component that has been selected in the Resource Tree view. It colours components according to their host and provides some basic indication of their current availability. System operators can select the icons representing components in the Dependency view in order to navigate along all discovered dependencies. The Resource Details view provides some basic information about each selected component, such as its host, component type and availability statistics.

System operators can inspect the metrics that have been observed for a component by dragging this component into the Selected Resource Metrics view (Figure 5.3). The view shows a tabular overview of all metrics that are available for this component. Values are updated as new measurements come in and, where suitable, data is aggregated. When a system operator double-clicks on any of the metrics a chart of its historical records is displayed (Figure 5.4).

The Application Activities view displays all activities performed by the application (i.e. the BPEL processes defining a service composition). For each recorded activity, its start and end times are given along with any additional details that are available. When a system operator selects a particular application activity, such as for example the invocation of a Web service, the UI will display only measurements and events that have taken place during a short time range around this activity. Finally, the Log view lists all error and warning messages that have been observed along with the component that reported this message.

We leave a description of the implementation of the dependency discovery process to Appendix A, but need to note the following about its current state. Our case study (Section 5.2) uses BPEL 1.1 to compose its various Web services. As this version of BPEL did not yet allow for the use of import elements, we have hard-coded the service-level dependencies that would otherwise result from the static analysis. However, a proof-of-concept of our static analysis method exists in the form of the so-called ActiveBPEL runtime plugin (ABR), which I have developed as a reference implementation to demonstrate that deployment of BPEL workflows modelled with the Eclipse BPEL Designer can be automated. Version 0.4 of

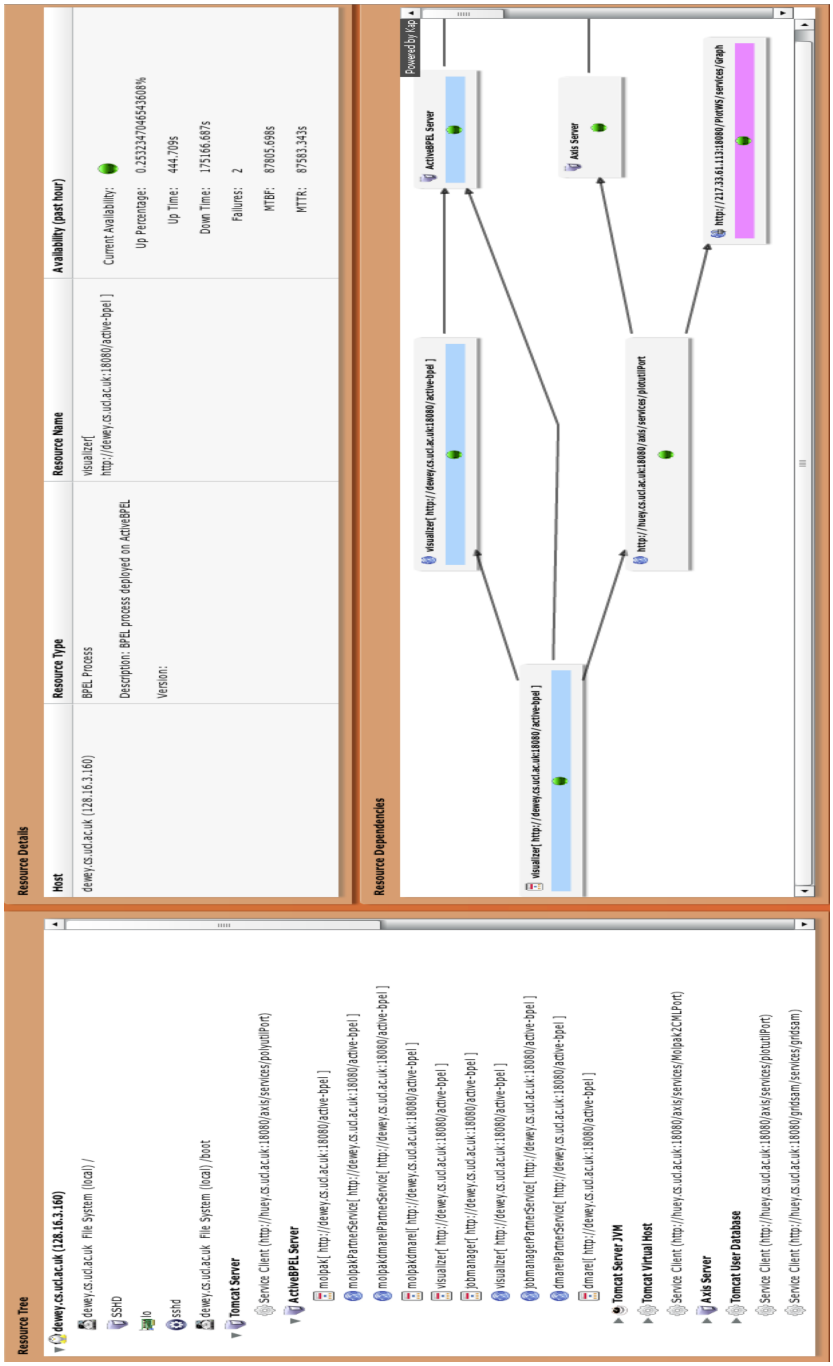


Figure 5.2: The Resource Tree view shows a hierarchy of all discovered components per host in a domain. The Dependency view displays dependents and dependencies for a selected component across hosts and administrative domains.

the ABR plugin is available from http://sse.cs.ucl.ac.uk/projects/past_projects/omii_bpel/downloads/ and the corresponding source code is maintained in the CVS repository of the Software Systems Engineering Group at UCL.

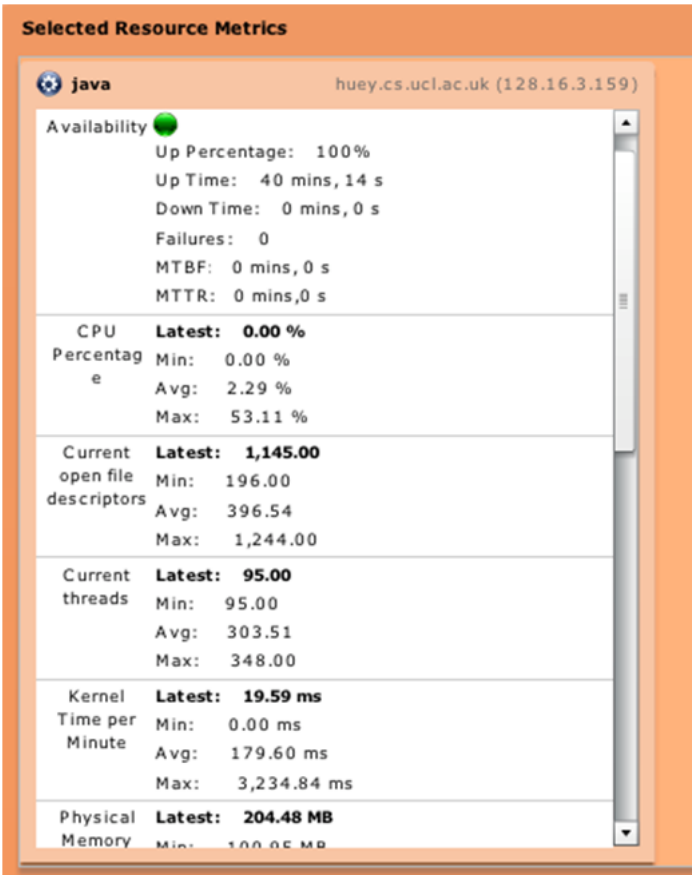


Figure 5.3: Metrics for a component can be inspected in more detail.

5.2 Evaluation Overview and Case Study

We begin with a high-level overview of the objectives of our evaluation and the corresponding experiments. We follow the guidelines on experimentation in software engineering as they are presented in [Wohlin et al., 2000] and accordingly have selected statistical tests from the ones suggested by [Wohlin et al., 2000] and by [Moore and McCabe, 2005]. We divide the evaluation of our approach to data-driven diagnosis into an intrinsic and an extrinsic one. The goal of the former is to demonstrate that our approach is feasible. An important aspect of its feasibility is the performance overhead it imposes on a system under observation. In order for our approach to be able to realise any benefits, its performance overhead needs to be limited. We perform a series of measurements of the Monere prototype applied to our case study in order to quantify the resource usage of the monitoring system and any slowdown it may cause to occur in the monitored service composition. We develop an analytical model of and measure the communication overhead that arises from the exchange of measurements within and across administrative domains. And finally, we present a brief analysis of the completeness and accuracy of the dependency graphs. The results of these analyses and measurements inform us about the feasibility of supporting data-driven diagnosis for large service compositions.

The extrinsic part of the evaluation is concerned with quantifying the effects of data-driven diag-

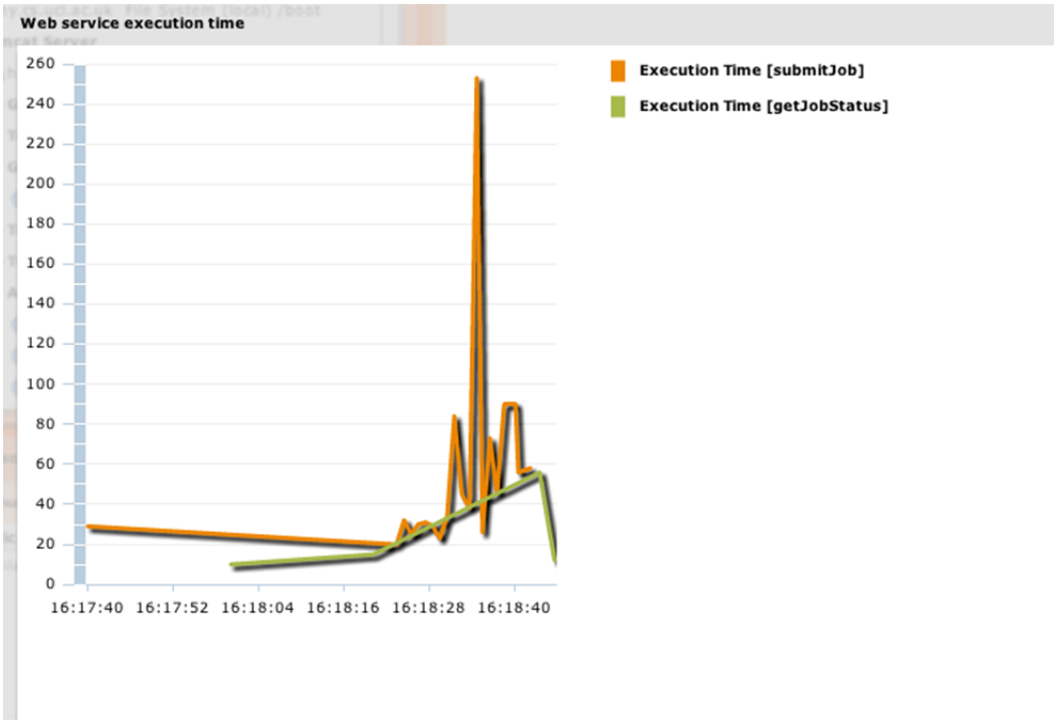


Figure 5.4: A chart depicting the development of an execution time metric for two Web service operations over time.

nosis and establishing its benefits. We want to find out whether data-driven diagnosis can improve the diagnosis of system-level failures in measurable ways. We quantify its effects based on two variables. We measure the diagnosis times and the success rate it can help system operators to achieve and compare their values to those achieved by system operators who engage in the standard diagnosis process we have described earlier. For this purpose, we perform a failure injection experiment in which we ask 22 participants to assume the role of system operators and to diagnose injected system-level failures. The participants are split between a control group using a standard tool set and another group using the Monere prototype for their diagnosis. We analyse the resulting data under various considerations and perform hypothesis tests to determine whether data-driven diagnosis can afford statistically significant improvements. The results of this experiment quantify the extent of the benefits of data-driven diagnosis and show us under what circumstances its application is justified and most beneficial.

Our case study is a cross-domain service composition that implements an application for the computational prediction of organic molecule structures. Organic materials can arrange the molecules they are comprised of in different ways along three dimensions. Each arrangement gives rise to a different crystal structure, which is also referred to as a *polymorph*. Each polymorph may result in different physical properties for a material. For example, polymorphs in a drug lead to different effective dosages. As there are many many different ways in which molecules can arrange themselves in three dimensions, it would be beneficial to computationally predict the most likely polymorphs for any given molecular

material. In Theoretical Chemistry, such computational prediction is based on performing an exhaustive search over molecule arrangements, also called *packings*.

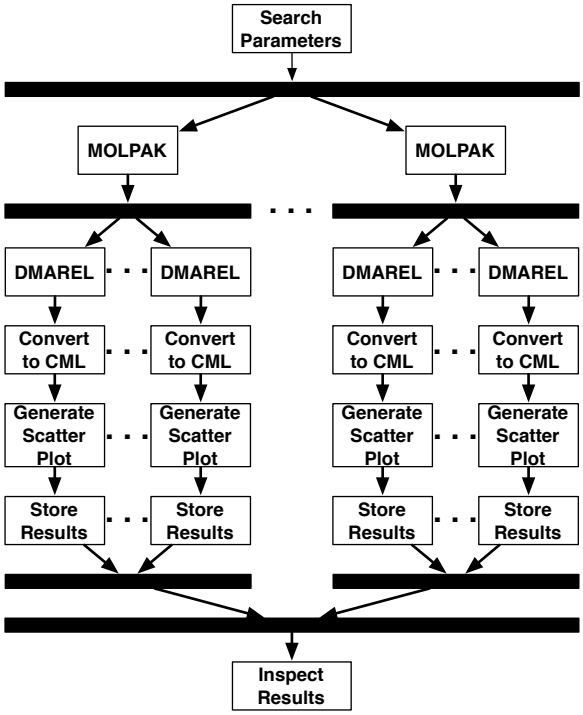


Figure 5.5: Overview of the Polymorph Search workflow for the computational prediction of organic molecule structures.

The Polymorph Search workflow in Figure 5.5 represents the process followed by a group of Theoretical Chemists at UCL for the computational prediction of crystal structures. The search is based on two Fortran programs that are commonly used in this application domain; MOLPAK [Holden et al., 1993] and DMAREL [Willock et al., 1995]. The former can generate up to 38 different potential molecule packings. The physical properties (e.g., density, energy) for each of these structures can then be calculated using DMAREL. Based on the physical properties, scientists can determine how thermodynamically likely a given molecule packing is. The precision of this calculation is based on the number of DMAREL instances that are executed on a packing. The maximum degree of precision that is supported is reached at two hundred instances of DMAREL per packing. To begin a new search, a scientist provides a description of the molecular material that is to be examined and sets other parameters, such as the desired level of precision. The search begins by executing a MOLPAK instance for each desired molecule packing. The output of each MOLPAK instance is then fed as input to up to two hundred DMAREL instances. The physical properties calculated by each DMAREL instance are converted into Chemical Markup Language (CML) [Murray-Rust and Rzepa, 1999] documents, which is a domain-specific format commonly used in this application domain. The results are also collected and rendered in a scatter plot and in a tabular format, before they are stored persistently for later inspection. Once this workflow

has completed, a scientist can then analyse the stored results to determine the likely polymorphs for the given material.

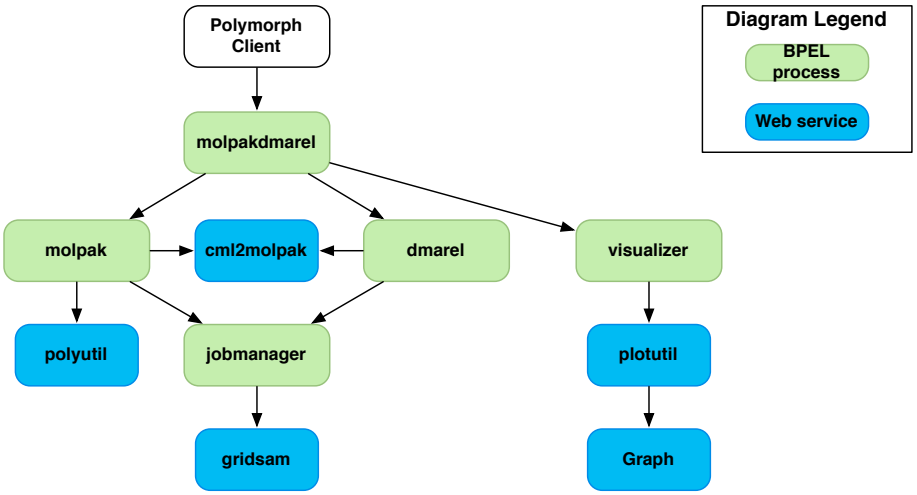


Figure 5.6: The Polymorph Search workflow as a BPEL service composition.

We have implemented this workflow as a BPEL service composition [Emmerich et al., 2005]. Figure 5.6 shows a simplified overview that omits the Web service interfaces of BPEL processes, XML Schema definitions and does not show how these artefacts are distributed. A web-based UI serves as the client through which scientists supply input data and start a new search. The workflow itself is hierarchically composed out of several BPEL processes. The top-level BPEL process, molpakdmarel, accepts the search input and coordinates among the other BPEL processes. It invokes one instance of the molpak process per requested molecule packing. Each instance converts the input data into a suitable format and makes use of the jobmanager process in order to schedule a corresponding instance of the MOLPAK program for execution on a Grid compute node. For each returned result from molpak, the top-level process invokes up to two hundred instances of dmarel, which, similarly to molpak, performs some transformations on the data it receives as input and then uses the jobmanager process to schedule DMAREL executions on a Grid. Finally, the top-level process invokes the visualizer BPEL process on the results. The visualizer process, in turn, stores the results in a tabular format and invokes a Graph service that renders them as a scatter plot. Both molpak and dmarel share a dependency on a Web service called cml2molpak that provides operations to convert data about molecule packings between different formats as needed during the various phases of the workflow. The molpak process furthermore relies on the polyutil Web service in order to store archives of analysis data and perform some clean up actions. The visualizer process uses the operations offered by plotutil, which prepares the resulting data to be rendered by the Graph service and converts it into a tabular format. Finally, the jobmanager uses the GridSAM Web service, which provides access to a Grid job scheduler. The Polymorph service composition is in active use by Theoretical Chemists at UCL.

We have deployed a clone of the production version of the Polymorph Search composition on a

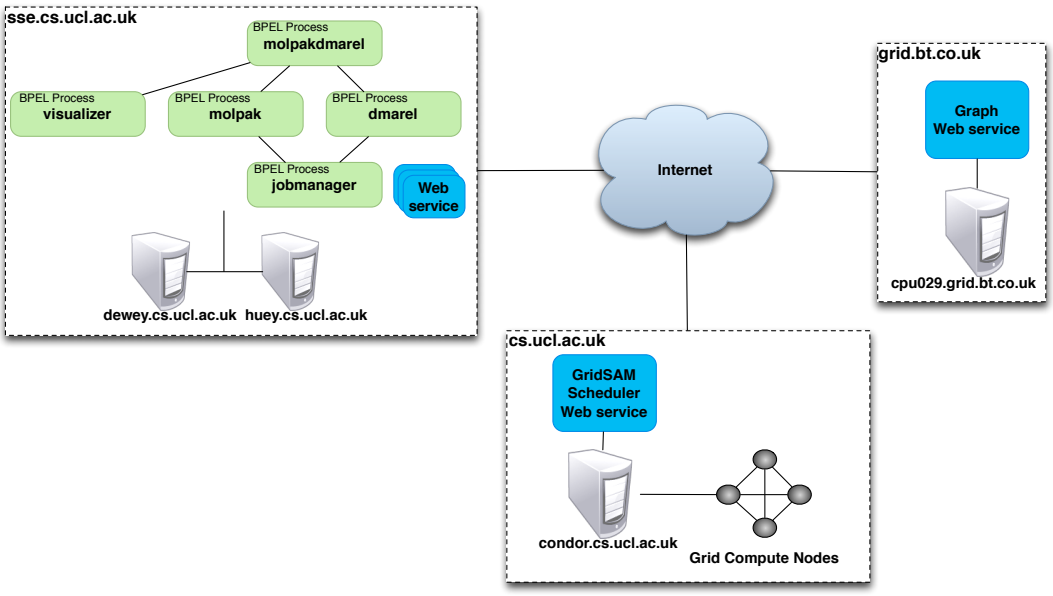


Figure 5.7: The deployment of the Polymorph Search service composition across administrative domains on our testbed.

distributed testbed as it is shown in Figure 5.7. The BPEL processes along with their various utility Web services and backend implementations are deployed across two hosts in the *sse.cs.ucl.ac.uk* domain. Access to 64 Grid compute nodes is provided through a Condor [Litzkow et al., 1988] instance on host *condor* in the same administrative domain. We did not have access to modify or install anything on host condor. Therefore, the GridSAM scheduler Web service and its GridSAM server are in fact deployed on host huey. We rely on a Condor client on host huey that queries the remote Condor instance about various measurements that Condor makes available and thus acts as a kind of proxy. We overcome the issue of access in this manner, and have chosen to depict the deployment of GridSAM as it would normally be. The Monere deployment in Figure 5.1 has agents deployed on two hosts (*huey* and *dewey* in the UCL domain) and the Monere server is on a third host, *louie*, that is not shown here. The Graph service is hosted by a separate administrative domain (*grid.bt.co.uk*). The production version of the Graph service is maintained and hosted by the University of Southampton. In order to be able to inject failures that affect the Graph service, we have deployed an instance of it on a server in a data centre maintained by British Telecom thus replicating the cross-domain dependency as it exists in the production deployment.

The Polymorph Search workflow is supported by a typical SOAP Web services stack as we have discussed it before (cf. Section 2.1.1). Figure 5.8 shows the entire stack, but only parts of it may be deployed on any given host. All hosts use some distribution of the Linux operating system. CentOS 5 [CentOS, 2012] is installed in the UCL domain and the hosts in the remote data centre use Fedora Core 8 [Red Hat, Inc.,]. We rely on Apache Tomcat [The Apache Software Foundation, 2012b] (version 5.0.28) as the application server in our deployment. It hosts instances of Apache Axis [The Apache Software Foundation, 2012a] (version 1.2.1) as the SOAP runtime, which in

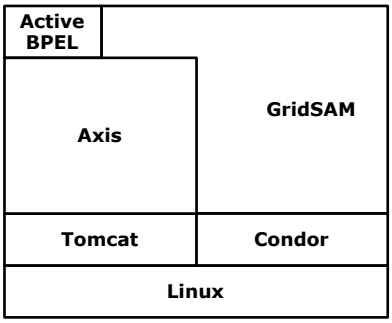


Figure 5.8: The middleware stack used for the Polymorph Search service composition.

turn hosts Web services and manages the communication between local and remote services. ActiveBPEL [Active Endpoints, 2012a] (version 3.0) serves as the BPEL runtime. It hosts and executes the BPEL processes comprising the Polymorph Search composition. ActiveBPEL depends on the services of Axis to communicate with partner services and it relies on Tomcat as its runtime environment that provides access to computational resources. Condor is a Grid job scheduler. It accepts scripts that define compute jobs in the form of executables, input data and optional constraints on their execution and schedules these compute jobs for execution on idle Grid compute nodes. The compute nodes mainly consist of both 32-bit and 64-bit desktop PCs in the student labs at the Computer Science department at UCL. GridSAM [Lee et al., 2004] manages the interaction between an application at the service level and an underlying Grid job scheduler. It accepts job descriptions in an XML-based format called the Job Submission Description Language (JSDL) [Anjomshoaa et al., 2005] and then uses the submitted JSDL documents to generate scripts for the underlying Grid job scheduler. The Web service interface of the GridSAM server provides operations for the submission of compute jobs as well as for their rudimentary monitoring to indicate whether a job is still in progress, has finished or has failed.

Our case study is somewhat simpler than the example application we introduced in Chapter 2, but it nevertheless shares many of the same characteristics. The Polymorph Search workflow is a middleware-based service composition and as such consists of numerous components that form dependencies on each other. The composition contains a cross-domain dependency from the visualizer BPEL process onto the Graph service, which is hosted by a different administrative domain over which system operators of the UCL domain have no control. Finally, the Polymorph Search composition is resource-intensive due to the demanding nature of its application domain. Each search produces and handles a potentially large amount of data (e.g., a single search can produce several GBs of data). Furthermore, the Polymorph workflow performs a large number of intensive computations on this data concurrently. These factors make the Polymorph Search workflow a suitable candidate for a case study based on which to evaluate our approach.

5.3 Performance Analysis

We begin by measuring the relative slowdown in the execution of a monitored application and examine whether it is statistically significant. Then, we measure the per-host overhead that arises from the resource usage of monitoring agents. And thirdly, we analyse and measure the communication overhead incurred from the exchange of measurements. There are two sources of overhead that we treat as static ones. The first is due to the need to maintain a dedicated host per administrative domain to act as the monitoring server. Furthermore, in its current implementation the discovery process takes less than 15 seconds for the discovery of 150 components across two hosts. Instead, we focus the performance analysis on the monitoring aspects of Monere.

The measurements are based on our online monitoring prototype, Monere, that is set up to monitor the Polymorph service composition. The composition is deployed on our testbed as it was shown in Figure 5.7 and Monere itself is deployed as was shown in its architectural overview in Figure 5.1.

5.3.1 Impact on Application Performance

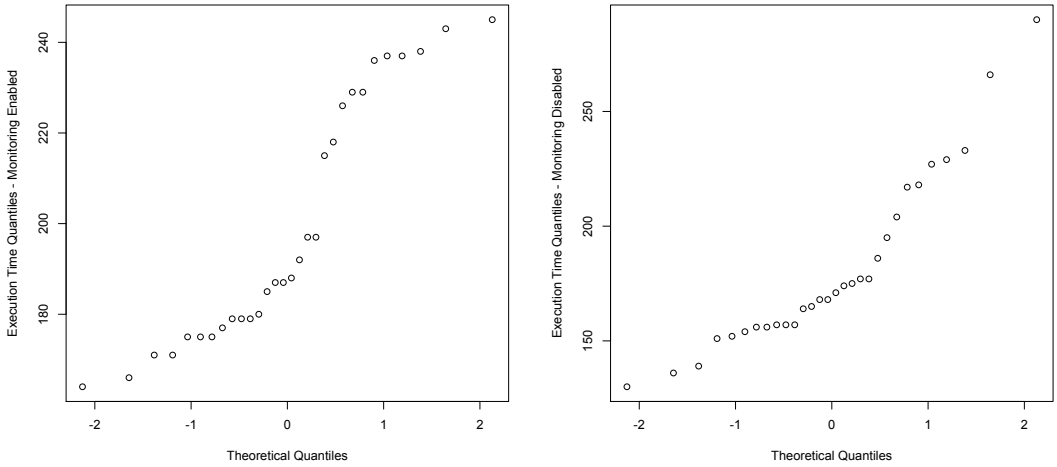
We evaluate the impact on the performance of an application under monitoring by measuring its execution time. In order to estimate the average slowdown to a monitored application, we have executed 30 searches of the Polymorph service composition without and then with monitoring enabled. We have done so on real input data sets as they are used by the owners of this application. The composition was configured to execute one instance of molpak and 19 instances of dmarel. This allowed us to repeat the measurements more often and to produce a larger sample size. The execution time was measured as the time from when a new search was submitted to the completion of the search. We relied on the start and end times that were recorded by the BPEL engine for the molpakdmarel process, an instance of which is created upon receipt of the initial input message and completes once all results have been stored persistently.

Table 5.1: Execution times of the Polymorph service composition with and without monitoring enabled.

	n	Mean Time	Std. Dev.	Slowdown
Enabled	30	199.23s	27.10	9.7%
Disabled	30	181.63s	38.31	

The results for each of the 30 searches is summarised in Table 5.1. We observe an increase in the execution time of the Polymorph service composition when it is under monitoring of 9.7%. The relatively large standard deviations can be explained due to differences in the time it takes the Grid scheduler to schedule jobs on the compute nodes, which depends on their utilisation.

We perform a two-sample *t* significance test in order to determine whether the observed difference in the mean execution times is statistically significant. Our null hypothesis states that there is no significant difference in the mean execution times of the two samples. The alternative hypothesis states that the difference is too large to be due to mere chance and that it is statistically significant (i.e. the significance level of our test is $\alpha = 0.05$). We refer to the mean of the execution times when monitoring is disabled



(a) Monitoring Enabled (b) Monitoring Disabled

Figure 5.9: Normal Quantile plots of the execution time measurements.

and enabled by μ_x and μ_y respectively. More formally, the hypothesis statements are as follows:

$$H_0 : \mu_x = \mu_y \tag{5.1}$$

$$H_a : \mu_x \neq \mu_y \tag{5.2}$$

We use a two-sided two-sample *t* test that is similar to Welch's *t* test [Welch, 1947] with the difference that we use the convention of using the smaller of $n_x - 1$ and $n_y - 1$ as the degrees of freedom to determine the *t* statistic, instead of letting software estimate this. Our choice represents a more conservative form of approximation of the degrees of freedom ([Leaverton and Birch, 1969], [Scheffe, 1970], [Best and Rayner, 1987]). Whenever possible, we choose this test in preference over pooled two-sample *t* procedures as is for example suggested in [Wohlin et al., 2000, page 99], because pooled procedures assume equal population variances, which is difficult to verify in practice. The main assumptions of our two-sample *t* test are that the measurement data are close to normal and that they do not contain outliers [Moore and McCabe, 2005, pp. 463, 493]. We evaluate the distributions of the measurement data for normality graphically by using a pair of normal quantile plots. Points that are close to a straight line indicate a normal distribution. The normal quantile plots in Figure 5.9 confirm that the two samples are roughly normal. In any case, the two-sample *t* test is robust against skewed data at large sample sizes ($n \geq 40$) and our combined sample size is 60. Using the $1.5 \times IQR$ rule, we find a single high outlier in the execution times without monitoring at 290 seconds. We consider such a measurement to be reflective of the normal variation of execution times. Furthermore, a single outlier out of 30 samples should not influence the outcome of the significance test so as to change our conclusion.

To be able to reject the null hypothesis, H_0, for a two-sided test the absolute value of the *t* statistic calculated from the samples needs to be greater than the critical value of the *t* distribution at the given significance level. In our case the following needs to hold: $|t_0| > t_{\alpha/2,dof}$, where dof refers to degrees

of freedom. Similarly, the *P-value* needs to be less than the significance level α[1]. The critical value of the *t* distribution at the 0.05 significance level with 29 degrees of freedom is $t_{0.025,29} = 2.045$. The *t* statistic from our measurements is $t_0 = -2.054$ and the *P-value* for the two-sided test is 0.049. As $|-2.054| > 2.045$ and as the *P-value* is less than the chosen significance level of 0.05, we can reject the null hypothesis in favour of the alternative one that states that the observed difference in execution times is not due to random variation, but that it instead represents a statistically significant difference.

The 95% two-sample confidence interval around the mean difference in execution times, $(\mu_x - \mu_y) = 17.6$ seconds, is $(0.08, 35.12)$. Therefore, even though we have evidence of a difference, we cannot arrive at a precise estimate of what the average difference is. Given the data, the average slowdown could range from being barely measurable to a maximum of about 19%. If we take the mean execution times as measured, we find an average slowdown of less than 10%. One could argue that this kind of slowdown is not high given that Monere continuously monitors in excess of 100 components and a few hundred metrics. However, neither is such an overhead negligible and therefore it needs to be weighed against any possible savings in downtime that our approach may help to realise.

5.3.2 Per-Host Overhead

We evaluate the overhead that a monitoring agent imposes on its host by measuring some of the computational resources an agent consumes on our testbed. We have observed two agents for 60 minutes, during which they monitored the operation of the Polymorph Search composition. The two agents each run within their own JVM on two hosts (huey and dewey) in the UCL administrative domain. The agents monitor about 340 metrics each. Some of the components monitored by agent 2 have only a small number of metrics, which results in the almost equal number of metrics monitored by both agents. Given the different collection intervals of these metrics, which, in our case study, range from five seconds to ten minutes, each agent takes about 300 to 400 measurements per minute. An agent can perform several measurements in parallel.

Table 5.2: The resource usage of two monitoring agents.

Agent	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

The resource usage measurements for the two agents are summarised in Table 5.2. The CPU on both hosts is a single-core Pentium 4 that is clocked at 3.2 GHz. We find that the CPU usage during the 60 minutes averaged at around 3.2% for agent 1 and at 4.2% for agent 2. The average amount of used heap space was 11MB and 18MB respectively. The resource usage of agent 2 is almost one third higher than that of agent 1 in in spite of monitoring approximately the same number of metrics.

[1] As a brief recap and to avoid confusion, *t* is the critical value of the *t* distribution at a given significance level. t_0 is the test statistic calculated from the samples. It measures how far the two means are apart. The *P-value* is the probability that the two sample means are as far apart as has been observed, if H_0 were true. The smaller the *P-value*, the more confidence we can have in rejecting H_0 and the less likely it is to observe a difference of such an extent.

The reasons of this are twofold. First, each component that an agent has discovered is represented by a Java object. Agent 2 needs to store almost one third more instances of these component objects than does agent 1. Furthermore, there is a computational overhead involved in taking measurements that is incurred per component. Agent 2 needs to iterate over a larger set of components and invoke the measurement interface of each to record new measurements for any of their metrics. Based on our observation of the two agents, it appears as if the resource usage of an agent grows approximately linearly with the number of components. The second reason for the increased resource usage of agent 2 is that it monitors a dependency on the Web service in the remote data centre. This involves some additional socket communication with the corresponding Monere Information Service (MIS) and, more importantly, requires the agent to parse the XML data that the MIS returns. Both of these demand some additional CPU time and memory.

Overall, if we consider the number of components and metrics that the two agents monitor, we find the resource usage requirements of an agent in terms of CPU time and memory to be relatively low, especially when compared to the capacities of modern computer systems. The number of components per host in our setup is not small for a middleware-based system. Should such agents need to be deployed in systems that operate at a larger scale, performance optimisations may allow them to maintain an acceptable resource usage.

5.3.3 Communication Overhead

We present a brief analysis of the communication overhead incurred by online monitoring with Monere and then report some measurements. The communication overhead in Monere can be split into a local and a global component. The local one arises from the exchange of measurement reports (MR) between the agents and the monitoring server within an administrative domain. The global communication overhead occurs across administrative domain boundaries. It consists of the traffic generated by the Monere Information Service (MIS) responding to requests for updates to measurements about the services in its domain by agents in other administrative domains.

First, we consider the global communication overhead. The size of a MIS MR message is fixed as it only contains measurements from the last 60 minutes. In our case study there are only two types of MIS MR messages; one reports the throughput and the other one the execution times recently achieved by a Web service. That is, each agent requests two MIS MR messages per minute and per remote Web service[2]. The message overhead experienced by a MIS increases linearly with the number of its unique clients and the number of its Web services monitored by these clients.

For the local communication overhead, we can assume that there will always be new measurements to report on within the 30-second report interval used by agents, given that many of the metrics are continuous. Therefore, the number of MR messages sent is directly proportional to the number of agents in a domain, with one measurement report being sent by an agent every 30 seconds. The size of a MR is a function of the collection intervals and the number of metrics an agent collects at these intervals.

[2]In the case study, agents rely on periodic HTTP HEAD requests to services in other administrative domains in order to record their availability. An HTTP HEAD request typically fits easily within a single packet.

Approximately, the size of MR data within an administrative domain for a single 30 second measurement period is given by

$$a \times \prod_{i=0}^{n} \frac{30}{c_i} \times |metrics_i| \times k, \tag{5.3}$$

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 in all, but very large-scale deployments.

We have performed measurements of the communication overhead incurred by Monere for our case study by capturing TCP packets with *tcpdump* [tcp, 2012] over a period of 30 minutes. The Polymorph Search composition was executing searches during that time. The bit rate from the exchange of MRs within an administrative domain was 3.06Kbits/s. This represents the communication overhead between the agents on hosts huey and dewey and the monitoring server on host louie. We found a bit rate of 0.48Kbits/s generated from the exchange of measurements across administrative domain boundaries by the MIS in the remote data centre and a single agent in the UCL domain. Our monitoring prototype has not been optimised for performance (i.e. it does not detect and collate duplicate values, etc.). However, these measurements suggest that the burden from online monitoring on modern networks is small.

5.4 Dependency Graph

We present a brief qualitative review of our dependency graph in order to better understand its strengths and limitations. The graphs our approach obtains are structural dependency graphs as opposed to functional ones. As such, all dependencies that are represented in our graphs do exist in the analysed service composition. Monere discovers structural dependencies as they are known to exist between different types of components. This is in contrast to probabilistic approaches that may include dependencies that do not in fact exist in the system. Our structural dependency graphs are fully accurate in that they do not contain non-existent relationships. Another strength of our approach is that it is able to integrate dynamic changes that occur at runtime by performing the dependency discovery periodically. However, it may not be suitable for highly dynamic systems, in which changes occur with high frequency. Finally, our approach achieves a good level accuracy without requiring instrumentation to observe system behaviour under varying workloads.

Our structural dependency graphs are subject to a number of limitations. The completeness of our graphs is fully dependent on the available model information. Our approach will fail to detect components for which such model information has not been provided. Furthermore, these models need to be prepared by someone who possesses familiarity with middleware-based systems. Then again, this knowledge is encoded in a modular manner on a per-component basis and can be reused without modification for different deployments. Third, our approach does not discover data dependencies between

components, such as onto some element of the file system. We will have an opportunity to observe the effects of this limitation in our experiments. In addition, our graphs fail to explicitly represent the dependency of a Web service onto the classes that form its backend implementation. The graphs only capture a dependency between a Web service and its service runtime (that would usually host the implementation). Fifth, our approach does not discover calling relationships at the operating and middleware levels that are not associated with configuration or deployment information. We rely mainly on static sources of information and to obtain visibility of such calling relationships would necessitate instrumentation to trace the exchange of messages and requests at runtime. And finally, the resulting graphs may contain relationships that become active only rarely in a given deployment.

5.5 Effects of Data-Driven Diagnosis

5.5.1 Experiment Setup

The extrinsic part of our evaluation is concerned with quantifying the effects of data-driven diagnosis in order to learn the extent of any benefits that it may provide. For this purpose, we have designed a controlled experiment that we have carried out with human participants. The participants adopt the role of system operators of the UCL administrative domain in our case study. We inject several system-level failures into the Polymorph Search composition and present a participant with a typical complaint describing the user-visible symptoms of this failure. The participants are then asked to diagnose the injected system-level failure by employing either data-driven diagnosis with the help of the Monere prototype or a traditional diagnosis process based on a standard set of tools.

The independent variable in this experiment is the *diagnosis method* that a participant employs for diagnosis. Its two levels are data-driven diagnosis and traditional diagnosis. Each participant uses only one of these two treatments and we refer to the two groups as “Monere” and as “Control”, respectively. Participants in the Monere group were only given access to the Monere prototype to perform their diagnoses and received a user guide explaining how to use the web-based Monere UI. The instruction materials for the Control group consisted of an overview of their tool set, which was comprised of Linux root shells onto the hosts in the local administrative domain (i.e. participants were given administrative control of huey and dewey), JConsoles that allow for the inspection of the local Tomcat JVMs, pointers to various log files and a brief explanation of Condor shell commands that can be used to inspect the state of the Condor Grid job scheduler. Control group participants furthermore had access to a web browser for use as they saw fit as well as to a session of the ActiveBPEL monitoring console, which offers an overview of the state of executing BPEL proceses in the form of log files and in a graphical process representation. Participants in both groups were presented with a brief overview of the Polymorph Search service composition and its deployment on our testbed. The instruction materials are reproduced in Appendix B.

We evaluate the data resulting from the experiment based on two outcome variables. These are the *success rate* and the *mean time to diagnose* failures that have been achieved by the participants in each group. The success rate is the proportion of correctly diagnosed failures out of all those presented to the

participants. Given a user complaint about the symptoms of an injected system-level failure, a participant is asked to perform an early root cause localisation. That is, the participant needs to find the component that is most likely responsible for the failure and furthermore identify what about the state or behaviour of this component appears to have contributed to the failure. This second part ensures that a participant has identified a valid and correct cause that explains the given failure. A success is only counted, if both these questions are answered correctly.

The time to perform such a diagnosis is measured as the interval between the communication of the user complaint and the presentation of a correct diagnosis by the participant. I have measured the time from presentation of a user complaint about failure until a participant was able to supply a successful diagnosis using a stopwatch. There is a time limit per failure of ten minutes. If a participant has not been able to correctly diagnose a failure within this time, then we count the diagnosis as failed, relieve the participant and move on to the next failure injection. The mean time to diagnose a failure is defined as the average of observed diagnosis times across the participants in a group: $MTT_{diag} = \frac{1}{n}\sum_{i=0}^{n} t_i$. Success rate and mean time to diagnose represent objective measures, based on which we can formally evaluate the following two sets of hypotheses.

The null hypothesis about the success rate states that the proportion of successful diagnoses is approximately equal regardless of the diagnosis method that has been used. The corresponding alternative hypothesis states that the success rate when using data-driven diagnosis is higher than that achieved using a traditional approach with statistical significance (i.e. at the $alpha = 0.05$ significance level). We refer to the success rate of a group by its sample proportion of successes, $\hat{p}$.

$$H_0 : \hat{p}_{Monere} = \hat{p}_{Control} \tag{5.4}$$

$$H_a : \hat{p}_{Monere} > \hat{p}_{Control} \tag{5.5}$$

Similarly for the mean diagnosis time, the null hypothesis states that any difference in the diagnosis times achieved by the participants is merely due to random variance in the repeated measurements. The alternative hypothesis claims that the mean diagnosis times achieved by participants engaging in data-driven diagnosis are shorter than those observed for the Control group with statistical significance (i.e. at the $alpha = 0.05$ significance level). We denote the mean time to diagnose failures by MTT_{diag}.

$$H_0 : MTT_{diag}(Monere) = MTT_{diag}(Control) \tag{5.6}$$

$$H_a : MTT_{diag}(Monere) < MTT_{diag}(Control) \tag{5.7}$$

The experiment was performed in a laboratory setting. The failure injections and diagnoses were performed on our replica of the production version of the Polymorph Search service composition that we have deployed on our cross-domain testbed. The composition was executed on real-world input data sets during the experiment sessions. The injection of failures was performed with a simple fault injection framework that we have developed for this purpose. Our fault injection tool is able to reliably reproduce various conditions that lead to system-level failures. Instead of merely provoking the symptoms of such failures in the form of fault messages or exceptions, the injections reproduce actual failure causes. For

example, our tool can simulate a set of clients that overload a Web service with concurrent requests or it can execute processes that cause a particular JVM to exhaust its heap space or the number of threads available to the JVM. As another example, it can quickly exhaust the storage space on a given file system. The source code for the fault injection tool is maintained in the CVS repository of the Software Systems Engineering Group at UCL.

We selected six failures for injection during the experiment which were chosen from among ones we have observed over time during the operation of the Polymorph Search composition in production use. These six 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

These failures are typical of what we consider as system-level failures. They cover both ones that cause the abnormal termination of our case study (Failures 3 and 4) as well as ones that lead to severely degraded performance (Failures 1, 2, 5, 6). All of them prevent the application from making progress and/or from completing successfully, unless they are detected, their cause is diagnosed and they are repaired by a system operator. The causes of these failures include both high workloads as well as failure effects that propagate across administrative domain boundaries. Our case study is a multi-tiered and fairly complex system, in which most failures take root in various places in the middleware and can propagate across layers (e.g., Failures 2 - 6). Typical user complaints about these failures include reports that the execution of a search has failed or that parts of the expected results are missing. For Failures 2, 5 and 6 users usually complain that a search takes unusually long to make progress. The same set of six failures were injected in random order for all participants.

We have performed the experiment with a total of 22 participants. Both the Monere and the Control group had 11 participants each, which leads to a balanced experiment design. The participants were volunteers from among the PhD students, post-docs and the faculty of several Computer Science departments. A small number of professional software engineers were also among the participants. We used a questionnaire at the beginning of each session to determine the level of experience of a participant with a set of relevant technologies. The interview took place only after the participant had been randomly assigned to one of the two groups in order to remove any opportunity of letting the outcome of the questionnaire influence the assignment of a participant. As we did not have a pool of volunteers available, we alternated the assignment to one of the two groups with each new participant.

The questionnaire through which we quantified the relevant experience of our participants is reproduced in Figure 5.10. The technologies it covers are *distributed middleware*, *Linux*, *SOAP Web services*,

Technology	Experience
Distributed Middleware Technologies (J2EE, Tomcat, JBoss AS, etc.)	a. No knowledge b. General understanding (know what it is, know how it works) c. Some industrial experience (less than six months) d. Industrial experience (give number of years)
Linux	a. No knowledge b. General understanding (know what it is, know how it works) c. Some industrial experience (less than six months) d. Industrial experience (give number of years)
SOAP Web Services	a. No knowledge b. General understanding (know what it is, know how it works) c. Some industrial experience (less than six months) d. Industrial experience (give number of years)
BPEL	a. No knowledge b. General understanding (know what it is, know how it works) c. Some industrial experience (less than six months) d. Industrial experience (give number of years)
Grid Computing	a. No knowledge b. General understanding (know what it is, know how it works) c. Some industrial experience (less than six months) d. Industrial experience (give number of years)

Figure 5.10: Questionnaire used to determine participant experience with relevant technologies.

BPEL and *Grid computing*. The possible levels of experience are *(a) no knowledge*, *(b) general understanding*, *(c) some industrial experience* and *(d) industrial experience*. Option *(b)* represents someone with only little practical exposure to the technology, possibly gained over the course of a hands-on tutorial. Option *(c)* applies to someone with limited practical experience of up to six months on a single project and finally option *(d)* denotes a participant who has worked with a technology for several years on multiple projects.

We have assigned numerical scores to the nominal experience levels in order to be better able to compare them. The resulting comparison is shown in Figure 5.11. Each bar represents the cumulative experience score of the participants in a group with a particular technology. The numbers on the bars give the mode. We find that the levels of experience across all participants in the two groups is well-balanced. Therefore, we can regard the level of experience as a controlled factor.

In summary, we have described a controlled experiment that examines a single factor with two

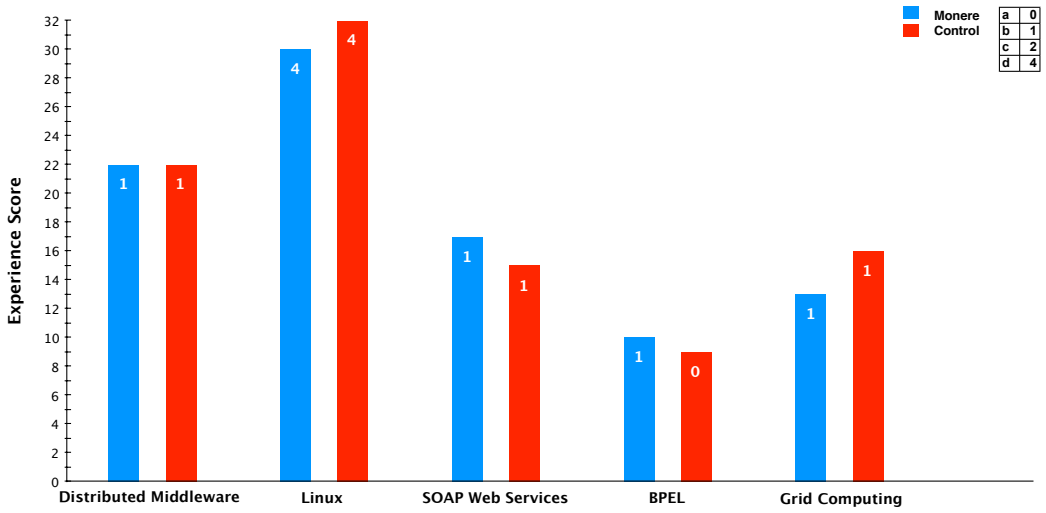


Figure 5.11: Comparison of the experience levels of participants in the effect and control group. The legend shows the assigned scores and the number on the bars denotes the mode.

treatments (data-driven and traditional diagnosis). The experiment takes place in a laboratory setting with the case study and six injected failures serving as the objects. The participants were split between an effect and a control group in a balanced way and the assignment of participants to groups and treatments occurred randomly.

5.5.2 Evaluation Method

The objective of our extrinsic evaluation was to determine the benefits of data-driven diagnosis based on online monitoring in terms of its impact on the success rate and mean diagnosis times that system operators can achieve when dealing with system-level failures. So far, the question of the utility of monitoring has only been addressed by argument or anecdotal evidence in the literature as we will discuss in the final chapter. In contrast, we wanted to provide the first evidence-based determination of its effects. There is a range of possible approaches that one could pursue to achieve this objective. Each approach differs in its ability to lead to valid insights.

In a field study, a phenomenon is observed in its actual context. In order to perform a field study in our case, we would have had to install the Monere prototype on a service composition in production use and observe the performance of its actual system operators on failures as they occur in the field. Ideally, we would then have compared their performance in terms of success rate and diagnosis times to an organisational baseline or to the performance achieved by system operators on a sister project, which uses a traditional approach to failure diagnosis.

The benefits of such an approach are obvious. The system operators are guaranteed to be sufficiently familiar with the system under observation and to be experts at the task of diagnosis. In addition, a field study enables us to observe a real system as it behaves under its natural workload with all the additional subtleties that may arise from day-to-day operation. However, field studies also have several

disadvantages. They offer little control over potentially confounding factors. And in order to enable us to observe a variety of failures, a project would have had to agree to participate in the study for several months. In general, field studies of this kind are expensive and difficult to arrange and it was therefore not within our resources to perform one. It can even be difficult to perform a field study with the resources of an industrial setting, given its cost and potential for disruption to production systems.

At the low end of the spectrum of available approaches a researcher might test the approach on herself. This usually results in an attempt to demonstrate plausible effects and benefits argumentatively. Furthermore, such a pseudo experiment could be based on a simplified, made up service composition and on failures from an analytic model instead of ones obtained on observation of real systems. These approaches have obvious shortcomings.

The controlled experiment we have described is a compromise between these two extremes. It achieves our objective through a rigorous empirical study of the effects of interest under as realistic a set of circumstances as is possible within a laboratory setting given our resources. It allows us to observe human participants as they perform failure diagnoses, although they are not expert system operators. We are able to perform the experiment on a replica of a real-world service composition and on failures as they have been observed during production use of that service composition. The use of a realistic system and set of failures limits opportunities to influence the outcome of the treatments and hence increases our confidence that we observe phenomena as they do in fact occur in the field. Furthermore, our experiment setup allows us to observe objective measures as our outcome variables. We counter the lack of a baseline of success rates and diagnosis times by comparing the outcome variables observed for participants in an effect group using Monere to those achieved by control group participants who use a standard set of tools as they are commonly used for failure diagnosis in distributed systems. In addition to being able to provide valid insights about the effect under study, an in-vitro study such as ours can be used as a precursor to a set of field studies.

5.5.3 Results

The following subsections present the results of our controlled experiment about data-driven diagnosis. Section 5.5.3.1 analyses the difference in successful failure diagnoses between the two groups and identifies the most frequently mis-diagnosed failures. Section 5.5.3.2 examines the diagnosis times achieved by the participants in each group for all failures and for some subsets of failures. Section 5.5.3.3 examines the diagnosis times of individual failures and in particular identifies the failures with the longest and shortest diagnosis times in each group. Finally, Section 5.5.3.4 considers the impact of experience levels on diagnosis times.

5.5.3.1 Success Rate

We begin with a comparison of the success rates achieved by the participants. We test our expectation that data-driven diagnosis leads to a significantly higher success rate than the traditional approach. We have expressed this expectation in the two hypotheses 5.4 and 5.5. In each group, 11 participants were presented with six failures each. This results in a total number of 66 trials per group. We have observed 63 successful diagnoses in the Monere group and 48 in the Control group. This translates to success

rates of 0.96 and 0.73, respectively.

We use a two-proportion z test [Moore and McCabe, 2005, p. 562] in order to determine whether the success rate of Monere participants is higher with statistical significance (i.e at the $alpha = 0.05$ significance level). The test is based on two assumptions. The first assumption is that the data have been obtained through independent random sampling. This condition is met by our experiment design, in which participants have been assigned to treatments at random and were presented with the same six failures in random order. We have obtained a combined sample size of 132. The second condition for use of this test is that the distribution of our sample data approximates the normal distribution. For this assumption to hold, the *expected* number of successes and of failures in each group needs to be at least 10 [Moore and McCabe, 2005, p. 344]. We determine the expected number using the pooled proportion estimate, $\hat{p}$, given that the null hypothesis 5.4 assumes equal population proportions.

$$\hat{p} = X_{Monere} + X_{Control}/n_{Monere} + n_{Control} \tag{5.8}$$

where X is the number of successes and n is the number of all trials. As the number of trials is equal in both groups, the expected number of successes for both is given by $n \times \hat{p} = 55.5$ and the expected number of failures by $n \times (1 - \hat{p}) = 10.5$.

We expect the success rate of participants using data-driven diagnosis to be higher than that of those participants using a traditional approach, as our alternative hypothesis 5.5 states. Therefore, we perform a right-sided test. In a right-sided test, we reject the null hypothesis, if the value of the z statistic, which we calculate based on our samples, is greater than the critical value of the normal distribution at the given significance level. That is, we need the following to be true: $z_0 > z_{alpha}$, where $alpha$ is the significance level. Similarly, the *P-value* needs to be less than the significance level. The right-sided critical value of the normal distribution is $z_{0.05} = 1.645$. The z statistic from our observations is $z_0 = 3.569$. The *P-value* for the right-sided test is 0.00018. As $3.569 > 1.645$ and as 0.00018 is considerably less than the significance level, we can reject the null hypothesis in favour of the alternative hypothesis which states that the success rate achieved by participants engaging in data-driven diagnosis is significantly higher that that of participants using traditional diagnosis.

Some statistical texts [Moore and McCabe, 2005, p. 562] suggest that five successes and failures is the minimum number required in order to be able to assume an approximately normal distribution for a proportion. As the number of observed failures in Monere is less than five, we have also performed a non-parametric test. The right-sided χ^2 test that we have used for this purpose returns the same *P-value* as our two-proportion z test and therefore confirms the conclusion we have reached.

If we look at the failed diagnoses more closely, we find that the most frequently misdiagnosed failures for the Control group are those affecting the Web service hosted in the remote administrative domain. These are failures 1 and 2, in which the remote Web service becomes unavailable or experiences a severe performance degradation. In the Monere group, we find that two out of the three failed diagnoses were due to failure 5, in which a component becomes unavailable due to the exhaustion of disk space on a file system volume. We will discuss these findings in more detail when we analyse the diagnosis times for individual failures in Section 5.5.3.3.

In conclusion, we find that a significantly higher success rate is achieved by participants when using data-driven diagnosis. The 95% confidence intervals are $(0.868, 0.989)$ for Monere and $(0.608, 0.820)$ for the Control group. Even though there are no available field studies with which we could compare these values, we find that Control group participants achieve a success rate of about two thirds to maximally four fifths, whereas Monere participants get almost nine of out ten diagnoses right on average. Furthermore, two thirds of unsuccessful diagnoses for the Monere group are due to just one of the six failures. This suggests that it could be possible to achieve a close to perfect success rate, if we were able to identify a common cause for these misdiagnoses and remove it from our approach or prototype.

5.5.3.2 Mean Time To Diagnose

Next, we examine the diagnosis times that participants were able to achieve. Our expectation is that participants who use data-driven diagnosis achieve significantly shorter diagnosis times on average. This expectation has been expressed in the two corresponding hypotheses 5.6 and 5.7. Each group consisted of 11 participants and each participant was asked to diagnose the same six failures in random order. There were 63 successful diagnoses for which we could record diagnosis times in the effect group and 48 diagnosis times have been recorded for the Control group. The outcome variable we use is the *mean time to diagnose* a failure as defined in Section 5.5.1.

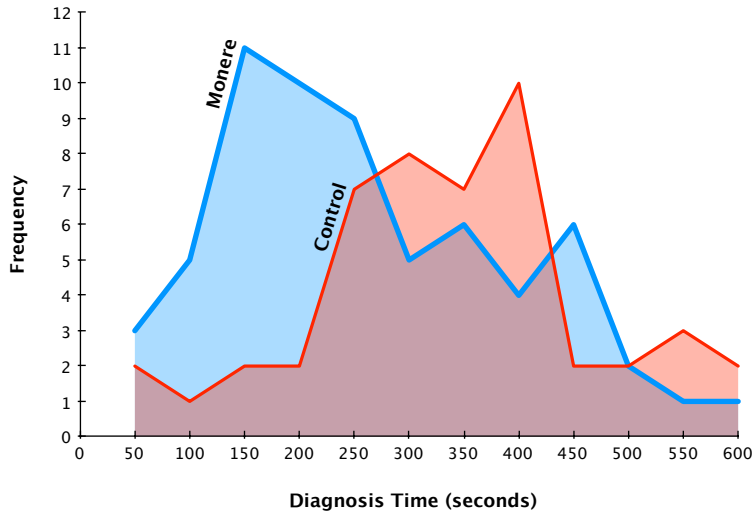


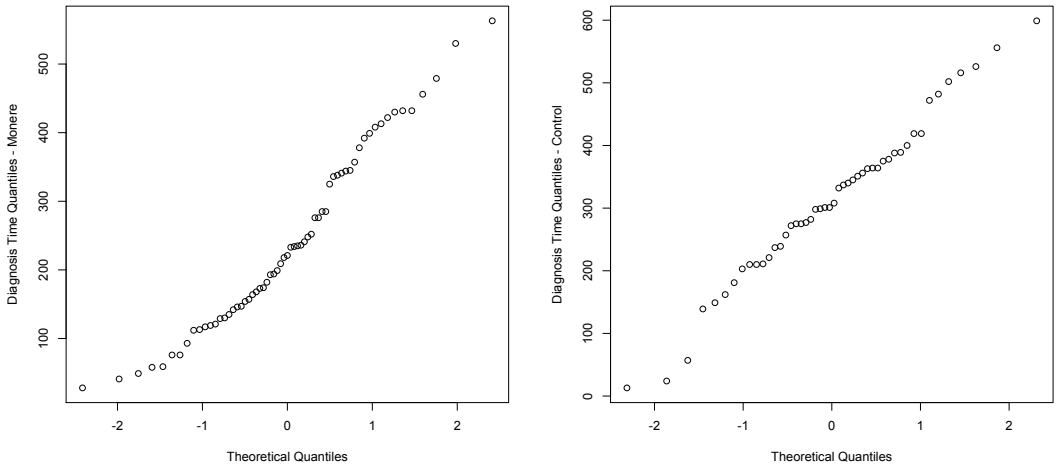
Figure 5.12: Histogram of the diagnosis times observed for participants in the effect and control groups.

The histogram in Figure 5.12 provides a graphical overview of our measurements. The x-axis groups the observed diagnosis times into buckets that are 50 seconds wide. The y-axis displays the number of observations that fall into each bucket. The histogram suggests that the participants in the Monere group who have used data-driven diagnosis have been able to diagnose failures more quickly than their counterparts in the Control group. However, the histogram also reveals a cluster of observations for the Monere group of around seven-minute diagnosis times. We need to quantify our observations in more detail to answer our research question.

Table 5.3: The MTT_{diag} observed for participants in the Monere and in the Control group.

	n	MTT_{diag}	Std. Dev.	Difference
Monere	63	241.56s	132.34	22.57%
Control	48	311.96s	129.71	

We have summarised the results in Table 5.3. The MTT_{diag} in the Monere group is 241.56 seconds. The participants in the Control group achieved an MTT_{diag} of 311.96 seconds[3]. The relatively large standard deviations are indicative of the normal variation of diagnosis times, which range from a few tens of seconds to almost ten minutes. The table confirms and quantifies what the histogram suggests. The participants who used data-driven diagnosis were able to diagnose failures more quickly. Across all six failures, there appears to be a reduction in diagnosis times of 22.57%.



(a) Data-driven diagnosis - Monere

(b) Traditional diagnosis - Control

Figure 5.13: Normal quantile plots of the diagnosis time measurements for all six failures.

We use a two-sample *t* test in order to find out whether the observed difference in diagnosis times is in fact of statistical significance. As we have already discussed in detail in Section 5.3.1, the test that we have chosen in preference over pooled two-sample *t* procedures is similar to Welch's *t* test [Welch, 1947] with the difference that we use the smaller of the two sample sizes minus one as a conservative approach to determine the degrees of freedom. The main assumptions of the two-sample *t* test are that the measurement data are close to normal and that they do not contain outliers [Moore and McCabe, 2005, pp. 463, 493]. The normal quantile plots in Figure 5.13 suggest that our measurements are normally distributed. Furthermore, the two-sample *t* test is robust against skewness at large sample sizes ($n \geq 40$). In this case, the combined sample size is 111. We defer a discussion of outliers until the next section, but we note here that, compared to the sample size, the number of outliers is sufficiently small so as not to affect

[3] We discuss the small number of outliers that exist in our data set in the next section.

the conclusion we reach based on the test.

We test our expectation (hypothesis 5.7) by performing a left-sided test. In order to be able to reject the null hypothesis, the value of the *t* statistic calculated from our samples needs to be less than the critical value of the *t* distribution at the given significance level and degrees of freedom. The decision rule for this test is as follows: $t_0 < t_{\alpha,dof}$, where α is the significance level and dof stands for degrees of freedom. Similarly, we require that the test's *P-value* be lower than the significance level $alpha$. The critical value of the *t* distribution at the 0.05 significance level with 47 degrees of freedom is $t_{0.05,47} = -1.678$. The *t* statistic from our measurements is $t_0 = -2.808$. The *P-value* for the one-sided test is 0.00362. As $-2.808 < -1.678$ and as the 0.00362 is less than the significance level of 0.05, we can reject the null hypothesis in favour of the alternative hypothesis. System operators who use our prototype to perform data-driven diagnosis do, on average, achieve significantly shorter diagnosis times than system operators who rely on the traditional approach using a standard set of tools.

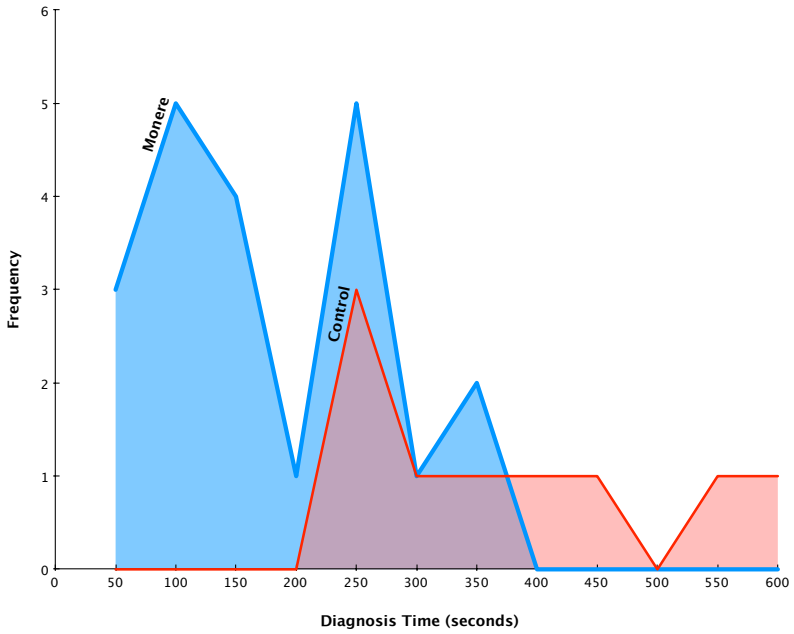
The MTT_{diag} observed for Monere is 70.40 seconds less than that observed for the Control group. This corresponds to an average reduction in diagnosis times of about 22.5%. The 95% confidence interval around the difference in MTT_{diag} between the two groups is $(19.969, 120.837)$ in seconds. That is, the true reduction in diagnosis times that data-driven diagnosis can afford across all six failures is somewhere between 6% and 38%. Even though this represents a relatively large margin of error, the data clearly shows a significant reduction in diagnosis times.

Next, we examine the difference in diagnosis times for two subgroups of failures that arise when we divide the six failures based on their locality. The cause of Failures 1 and 2 lies in the remote administrative domain. They are failures of the Graph Web service, which is hosted at the remote data centre and which the Polymorph service composition integrates into its workflow. The remaining four failures all affect components in the local (UCL) administrative domain. The mean experience scores of participants who have contributed to our diagnosis time observations for remote failures are approximately the same in both groups (Monere = 8.36, Control = 8.86). For local failures, all participants have contributed to our observations and hence the levels of experience remain well-balanced among both groups as discussed before. We proceed as before in order to determine the effect for these two groups of failures.

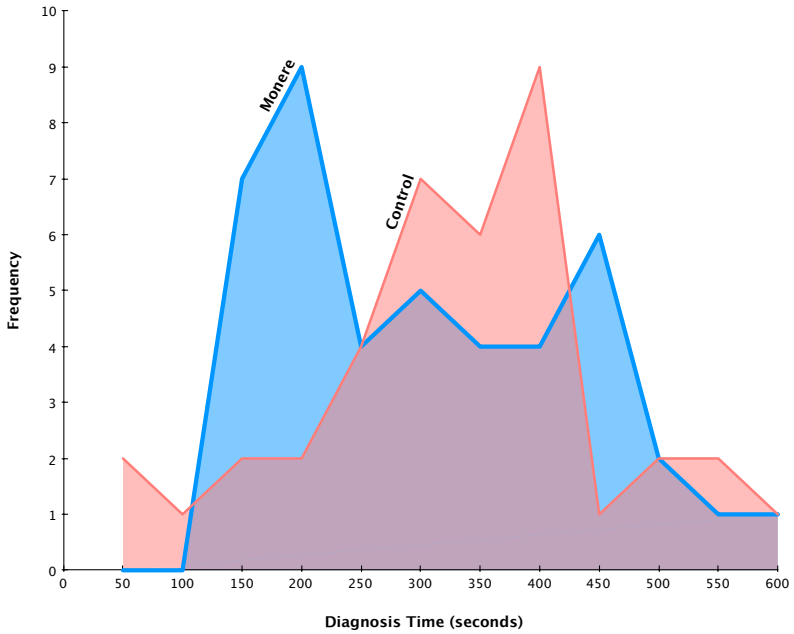
Table 5.4: The MTT_{diag} for failures in the local and a remote administrative domain.

	n	MTT_{diag}	Std. Dev.	Difference
Monere Remote	21	156.62s	95.71	55.29%
Control Remote	9	350.33s	133.12	
Monere Local	43	283.26s	126.92	6.55%
Control Local	39	303.10s	129.03	

Figure 5.14 displays two histograms that compare the distributions of observed diagnosis times for each group of failures. There appears to be a much larger proportion of shorter diagnosis times in the Monere group for failures whose cause lies in the remote domain (Figure 5.14(a)) than was the case



(a) Diagnosis times for failures in the remote domain.



(b) Diagnosis times for failures in the local domain.

Figure 5.14: Histogram of diagnosis times for failures in the remote and local administrative domain as observed for the effect and control groups.

when considering all six failures. However, the second histogram in Figure 5.14(b) does not reveal such a pronounced advantage of data-driven diagnosis for failures in the local domain. If we were to judge the effect solely based on the visible area under the curve, the reduction for local failures would appear to be less than what we were able to observe for all six failures.

Table 5.4 further quantifies and summarises our results. When applied to failures in the remote domain, data-driven diagnosis results in an MTT_{diag} of 156.62 seconds, which compares favourably with a time of 350.33 seconds observed for participants in the Control group. This corresponds to an average reduction in diagnosis times of 55.29%. The difference in diagnosis times for failures whose cause lies entirely within the local administrative domain is much smaller. For the Monere group we find an MTT_{diag} of 283.26 seconds, which is only 20 seconds less than the time we have observed for the Control group. This represents a mere reduction in diagnosis times of 6.55% when using data-driven diagnosis for failures in the local administrative domain.

Again, we want to determine whether these differences are statistically significant. The sample sizes of observations for failures in the remote administrative domain are rather small. This makes it difficult to assess the data for normal distribution[4]. Therefore, we use the *Mann-Whitney-Wilcoxon* test [Mann and Whitney, 1947] as a non-parametric alternative. Its main assumptions are that the observations in the groups are independent of each other and their measurement scale is at least ordinal.

We perform a one-sided (left-sided) test at the $\alpha = 0.05$ significance level. In order to reject the null hypothesis in favour of the alternative one that states that data-driven diagnosis helps to realise a reduction in diagnosis times, we need the *P-value* from the test to be less than α. The *P-value* for the remote case is 0.000405 and for the local case we find it to be 0.2017. For the remote case, we can clearly reject the null hypothesis. However, whilst data-driven diagnosis clearly leads to significantly shorter diagnosis times when applied to the diagnosis of failures whose cause lies in another administrative domain, the same cannot be said when system operators apply this approach to deal exclusively with local failures. In the latter case, we have not found sufficient evidence to enable us to reject the null hypothesis.

We quantify the observed differences in MTT_{diag} in a bit more detail. For failures in the remote domain, data-driven diagnosis has enabled participants to spend 193.71 seconds less on diagnosis than participants in the Control group did. This corresponds to an average reduction in diagnosis times of about 55.29%. The 95% confidence interval around this difference is $(80.620, 306.809)$ in seconds, which corresponds to a range of diagnosis times reductions of between 23.01% and 87.58%. In contrast, for the diagnosis of failures in the local domain, the average difference is only 19.85 seconds; a reduction of 6.55%. The confidence interval is $(-37.465, 77.158)$, which, at best, results in a reduction of 25.46%, but can also translate into an increase in diagnosis times of slightly over 12%.

In summary, we find that data-driven diagnosis enables system operators to achieve significantly shorter diagnosis times for typical system-level failures. This is particularly true for cases in which the cause of a failure lies in another administrative domain. We estimate the average reduction in diagnosis

[4] At small sample sizes, we cannot rely on visual inspection to reveal the distribution of our data and formal tests of normality have only little power. Similarly, as the Central Limit Theorem is not in effect, parametric tests may provide inaccurate results.

times in such cases to be about 55% with a margin of error of approximately 32%. Across all six failures, we are again able to confirm a statistically significant reduction for data-driven diagnosis with a mean of about 22% and a margin of error of about 16%. Finally, the picture becomes less clear when data-driven diagnosis is applied to failures whose cause is entirely contained within the local administrative domain. If data-driven diagnosis was to be applied exclusively to local failures, then our data suggests that its use offers no discernible improvement over the traditional approach.

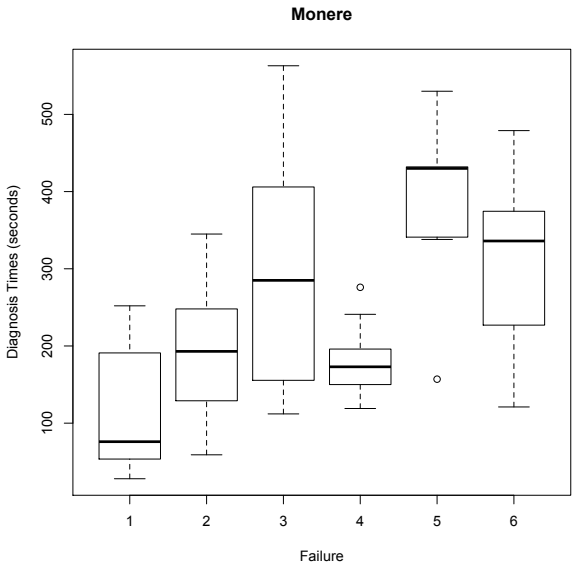
This raises the question whether data-driven diagnosis is in fact primarily of benefit for failures that originate in other administrative domains. Based on our data, this is an interesting question. After all, the confidence interval around the MTT_{diag} for local failures is not symmetrical around 0. Instead, a larger proportion of its values lie to the right of 0, which points to a tendency for reduced diagnosis times. We analyse the diagnosis times of individual failures next in order to clarify this question.

5.5.3.3 Failures

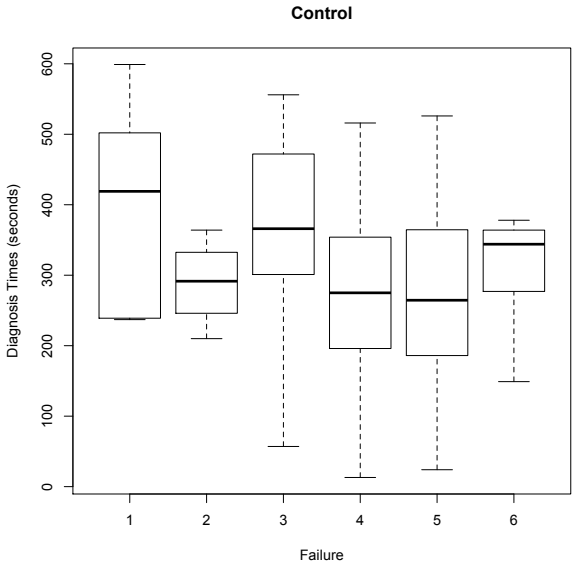
An analysis of the diagnosis times for individual failures in each of the two groups leads to a number of additional insights. Figure 5.15 displays box plots of the diagnosis times for each failure. There are two data points for Monere that are further than $1.5 \times IQR$ away from the median. These data points form one high outlier for failure 4 (heap exhaustion) with a recorded diagnosis time of 276 seconds and a low outlier for failure 5 (Condor unavailable due to disk space exhaustion) with a diagnosis time of 157 seconds. We have not removed the outliers from the dataset as we could not determine anything unusual about the participants who produced these diagnosis times nor with their experiment sessions. These observations are probably due to chance, which apparently plays some factor during the search for failure causes. Occasionally, a system operator happens to come across a conclusive or helpful piece of information early in the diagnosis process, while at other times she may spend more time on eliminating less helpful clues about the behaviour of a system first.

One observation that we can make based on the box plots is that the median diagnosis times for individual failures appear to be more similar in the Control group than is the case in the Monere group. In particular, it seems that participants in the Monere group had difficulty with diagnosing failure 5 in a timely manner. The box plots also reveal that failure 4, which is a local failure, has relatively short diagnosis times. Its median diagnosis time is even less than that of failure 2, which is one of the two remote failures.

In order to clarify the question that has been left open at the end of the last section, we compare the MTT_{diag} per failure in Table 5.5. Negative values of ΔMTT_{diag} denote reductions in the time to diagnose achieved by Monere participants. The first observation we make is that data-driven diagnosis leads to reductions in diagnosis times for almost all failures and not only for the remote failures (i.e. Failures 1 and 2). As we have already noted, Monere participants appear to have had difficulty diagnosing failure 5 in a timely manner. If we regard failure 5 as an exception and ignore it for now, we are better able to determine the effect of data-driven diagnosis on local failures. We compare the contribution made to the observed reduction in MTT_{diag} from the remote and from the remaining local failures (i.e. Failures 3, 4, 6). The proportion of the reductions is about 68% for the remote failures and 32% for the local



(a) Failure diagnosis times for Monere.



(b) Failure diagnosis times for Control.

Figure 5.15: Box plots of diagnosis times per failure in each group.

Table 5.5: A comparison of the Mean Time To Diagnose (in seconds) for each failure. Negative differences indicate a reduction in diagnosis times achieved by Monere participants.

	Monere	Control	ΔMTT_{diag}
Failure 1	121	399	-278
Failure 2	196	289	-93
Failure 3	283	353	-70
Failure 4	179	273	-94
Failure 5	388	272	116
Failure 6	304	311	-8

ones. We can conclude that while data-driven diagnosis can provide more substantial benefits in terms of reduced diagnosis times for failures whose cause lies in another administrative domain, its benefits are also realised to a large extent for system-level failures in general. The last hypothesis test from the previous section fails to show this clearly as the observed times for Failure 1 skew the observed effect in favour of remote failures and the diagnosis times of failure 5 does the opposite for local failures.

We also find that the diagnosis times in both groups for Failure 6, which represents the overloading of the Condor Grid job scheduler, are quite similar. Condor offers a set of shell commands that make necessary metrics available. In this case, Monere has no opportunity to provide additional data or aggregations on top of what Condor already makes available. This most likely explains the similar diagnosis times in both groups.

The table reveals remarkable diagnosis times for Failures 1, 5. We can explain this by identifying and discussing the shortest and longest failures in each group. As Table 5.6 reveals, the failures with shortest and longest MTT_{diag} are exactly reversed between the two groups. In the Control group the failure with the longest MTT_{diag} is Failure 1 (remote service unavailable) with a time of 399.20 seconds and the shortest times have been observed for Failure 5 (Condor unavailable due to disk space exhaustion). This is exactly reversed for the Monere group, as Table 5.6 shows.

Table 5.6: The failures with shortest / longest MTT_{diag}.

	Control	Monere
Longest	Failure 1 (399.20)	Failure 5 (388.22)
Shortest	Failure 5 (272.50)	Failure 1 (120.73)

This observation can be explained as follows. The relatively long diagnosis times for Failure 1 are due to a lack of adequate tool support that provides any visibility of critical components within separate administrative domains. This leads Control group participants to examine many components in the local infrastructure. Only after not finding any problems that could help explain the problem, do participants have sufficient confidence to focus their attention on the remote dependencies. It is not clear why this does not similarly affect diagnosis times for Failure 2 (slowdown of remote service). Control group participants identified Failure 5 relatively quickly. Upon issuing one of the available Condor commands,

they would immediately learn that the Condor process appeared to be down, which prompted them to inspect its log file and based on that examine the available disk space. Monere participants had much more difficulty with identifying this problem as the available dependency graph did only reveal a dependency between Condor and the operating system, but not a direct one onto the file system. This left participants with a relatively large 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.

In Section 5.5.3.1 we have hypothesised that there may be a common cause for the mis-diagnoses we have observed for Failure 5 in the Monere group. The incompleteness of our dependency graph (i.e. omission of data dependencies) may in fact be this common cause for mis-diagnoses and long diagnosis times. By extending our dependency model to also represent data dependencies, as has been proposed in [Magoutis et al., 2008], we could remedy this deficiency and probably realise a more significant reduction for system-level failure whose cause involves data dependencies.

In summary, while data-driven diagnosis is particularly useful for failures whose cause lies in another administrative domain, its positive effect on a broad range of system-level failures is likely to be more significant than the last hypothesis test in the previous section suggests. The analysis in this section has also highlighted both the strengths and limitations of our dependency graphs to assist system operators with identifying failure causes. Finally, we have made an observation about the potential effect of software components that have been designed to supply a variety of metrics about their operation at runtime. Such components support system operators to engage in data-driven diagnosis and can lead to shorter diagnosis times for failures that involve them without requiring additional tool support. However, judging from experience, such components are quite rare in practice.

5.5.3.4 Impact of Experience

Finally, we want to find out what impact the level of experience of a system operator may have on the diagnosis times she is able to achieve. For this purpose, we have calculated the correlation coefficients between the experience scores of participants and the diagnosis times that have been observed for them. If the correlation coefficient, r, was close to -1, then this would indicate a strong negative linear relationship. That is, in such a case a high level of experience would appear to be well correlated with lower diagnosis times.

Table 5.7: Correlation coefficients of the association between experience levels and diagnosis times.

	Overall	DS	Linux	WS	BPEL	Grid
Monere	-0.109	0.124	-0.323	0.141	-0.005	-0.342
Control	-0.037	0.195	-0.536	0.020	-0.017	0.178

The correlation coefficients given in Table 5.7 do not appear to reveal a strong linear relationship and scatter plots we have drawn do not suggest any other kind of association between experience and diagnosis times. No correlation is discernible across all technologies. While the highest diagnosis times are in fact produced by some of the participants with lower levels of experience, this is not true in all

cases. The same applies to experience with distributed middleware technologies, SOAP Web services, BPEL and Grid computing. Given that the failures affecting Condor only require a basic understanding of what a Grid job scheduler should do, the latter finding is not that surprising. However, more surprising is that knowledge of the service composition languages and distribution technologies is not required to perform well with diagnosing system-level failures. We find a slightly stronger negative correlation for experience levels with Linux. This effect is more pronounced for the Control group. This is likely due to the fact that participants in this group had to primarily rely on using Linux in order to perform failure diagnosis. And regardless of the assigned group, participants require a basic knowledge of the metrics that describe the operation of a distributed software system.

Contrary to what we would have expected to find, based on the available data, higher levels of experience do not appear to be well-correlated with shorter diagnosis times. However, our analysis of this impact should be viewed rather sceptically for two reasons. First of all, in order to achieve a better distribution of experience levels, we would require a much larger number of participants. Second, even though we have assigned numerical scores to the experience levels determined through the questionnaire, this remains a subjective measure that depends on how honestly a participant assesses her skills.

5.6 Threats to Validity

5.6.1 Conclusion Validity

Threats to the *conclusion validity* of an experiment may negatively impact the ability to reach valid conclusions. First, we consider the number of data points. Access to a greater number of participants would have increased the strength of our conclusions and reduced the margin of error. However, 111 observations of successful diagnoses across both groups is not an insignificant number. Even though we report relatively large margins of error at the 95% level, we are able to reach clear conclusions based on the significance tests in all cases but one. We responded to the relatively small number of observations of diagnosis times for remote failures by applying statistical methods that are suitable in such circumstances. We were unable to reach clear conclusions with regard to the impact of experience levels on diagnosis times. Here, the number of participants did have an impact on our ability to reach valid conclusions and so we have noted that a larger number of participants would be required in this case in order to increase our confidence in having observed a true effect.

Another threat to the conclusion validity can arise from the use of statistical tests that are not suitable given the data and evaluation objectives. We have tested our data to ensure that it meets all necessary conditions imposed by the significance tests we have employed. We have applied appropriate parametric and non-parametric tests as required by the characteristics of our data. Furthermore, we have approached the evaluation in an objective manner and have let our data analysis and tests inform our conclusions. Where appropriate, we have provided statistical summary measures.

The measures used in the experiment are another factor that can affect the conclusion validity. Most of the measures we have used are objective ones. We have explained how success rates and diagnosis times were measured. However, the experience levels and corresponding scores are necessarily sub-

jective measures to some extent. Our approach to mitigate potential threats arising from this, was to base the assignment of experience levels on the amount of time a participant has had experience with a technology.

Finally, the experiment design and its execution made adequate use of randomisation and of blocking. Participants were assigned to one of the two groups at random, before having answered the questionnaire. Each participant was presented with the same six failures in random order. The overall experience levels of participants appear to be well balanced among the two groups, which helps to control any potential effects this factor may contribute. No other influencing factors become immediately apparent.

5.6.2 Internal Validity

The *internal validity* of an experiment concerns the degree to which the independent variable is actually responsible for the observed effect on the dependent variable. Each participant encounters each failure only once and the order of their injection is determined randomly. Therefore, there is little opportunity for the order of failures to cause an effect. In addition to this, each participant only uses one of the two approaches. This reduces the likelihood that learning can occur during the experiment. A participant may improve her ability to debug the system over the course of an experiment session[5], but this should not influence the overall outcome in a significant way, given that participants are actively engaged in failure diagnosis for about 60 minutes.

Two other elements to consider are the instrumentation used to measure the outcome variables and the potential for the instruction materials that participants were provided with to have an effect. We measured the success rate by counting the number of successful and failed diagnoses according to the criteria outlined in Section 5.5.1. We used a stopwatch to measure diagnosis times. While we can expect the instrumentation to have worked accurately, the situation could potentially be different for the instruction materials. They may slightly bias the results in favour of the Control group, whose participants could have used it as a script to reduce the relevant problem space by focussing only on the components mentioned in the instructions. We have not observed such behaviour. However, even if it had occurred, this would only serve to increase our confidence that the observed difference in success rates and diagnosis times is due to a real effect of the diagnosis method that was applied.

Given that our participants were volunteers, one might expect them to be somewhat more motivated than the general population of system operators. However, in our experiment design we compare the performance of the two groups of volunteers with each other, instead of with a baseline obtained from a sample of the general population. Any effect from increased motivation can be expected to exist in both groups. In addition, as already stated, the levels of experience in both groups were well-balanced. Finally, participants in both groups were incentivised to take part in the same manner by being granted access to sweets during the experiment session.

5.6.3 Construct Validity

The *construct validity* of an experiment describes the extent to which the treatments and the outcome variables represent the real-world cause and effect under investigation. In our experiment the treatments

[5] We have not observed such an effect.

are data-driven diagnosis and the traditional approach to diagnosis. The former is represented by our prototype that provides additional information about system structure and behaviour. The latter treatment is based on a tool set as it is commonly used for system administration and debugging. No interaction between the two treatments can have occurred as each participant only took part in a single treatment. As already discussed, randomisation was used and the level of experience has been neutralised as a potentially confounding factor. Furthermore, failures were always presented in the same way by communicating a corresponding user complaint without offering any further guidance. The treatments represent the cause construct under investigation well and under the conditions in which they have been applied, it is safe to expect that what we measure is the effect of the two treatments.

Two of the primary effects of any approach to failure diagnosis are its impact on the number of successful diagnoses and the diagnosis times. A simultaneous increase of the former and reduction of the latter would contribute to higher system availability. Success rate and diagnosis time appear to be obvious choices for outcome variables that represent the effect we are interested in measuring for our treatments.

Our experiment is subject to mono-operation bias. We observe the diagnosis of system-level failures injected into a single service composition. It would not have been feasible, given our limited resources, to implement and deploy another composition of substantial dimensions and there is little reason to expect toy examples to lead to valuable insights. However, the Polymorph Search composition is a real-world application of moderate size and complexity. It exhibits many of the characteristics that we have identified in Chapter 2 in that it is a cross-domain, middleware-based and resource-intensive composition of several Web services. Therefore the Polymorph service composition is a promising object for our experiment. Nevertheless, it would be an interesting contribution to repeat our experiment on additional large service compositions in the future to see whether our results are confirmed.

5.6.4 External Validity

The *external validity* of an experiment describes how well its results generalise to a wider context. As we have explained, we had to rely on a single service composition. In addition, we had to limit the number of failures to six. It was not feasible to use more failures or considerably more difficult ones as most participants could not commit to more than 75 minutes. However, the Polymorph service composition is a real-world application of moderate size and complexity and has been executed under real input data sets. It encompasses the key system characteristics we are interested in. The failures we have selected for injection were from among a set of system-level failures that have been observed during production use of the Polymorph service composition. The selected failures are not trivial to diagnose given only typical end user complaints. Furthermore, the failure injection method we employ does not merely simulate the symptoms of a failure, but it reproduces the actual conditions in the system that lead to it.

None of our participants are actual system operators. However, all of them are Computer Scientists and software engineers with varying degrees of development and debugging experience. They could probably work successfully as system operators. Furthermore, we have compared the effect achieved in groups with equal experience levels. Nevertheless, the results could change to some extent, if the

experiment was to be repeated with participants who are considerably more experienced with diagnosing system-level failures in distributed systems.

While some of these factors are bound to limit the generalisability of our results to some extent, the experimental setup resembles a realistic setting as closely as is reasonably possible within a laboratory setting. The deployment of a real-world case study on a testbed into which we could inject actual system-level failures increases the trust we can have in the representativeness of our results. Furthermore, our results represent the first evidence-based determination of the effects of data-driven diagnosis and as such can form a basis for further experiments, such as field studies, in the future.

5.6.5 Repeatability

The final question we consider is whether or not our experiment is *repeatable* so as to enable independent verification of our results. We have described the experiment setup in detail. We have characterised the participants and have given an overview of the failures we injected. The source code of the failure injection tool and of the Monere prototype are available from the CVS repository of the Software Systems Engineering Group at UCL. The instruction materials used in the experiment are given in the appendix of this thesis. We are not at liberty to distribute the Polymorph service composition, but this does not preclude a repetition that uses another service composition. All this provides an adequate basis for repetitions of the experiment and may potentially inform a field study, in which a tool set that supports data-driven diagnosis could be installed on a real-world project in order to observe the performance of experienced system operators.

5.7 Discussion

The first question we set out to answer as part of our evaluation was whether our approach is feasible. In addition to the development of the Monere prototype, we have approached this question through a performance analysis to characterise various overheads that arise from the kind of monitoring that is necessary to support data-driven diagnosis. As part of this we have measured the slowdown in the execution time of a monitored service composition, the resource requirements of the monitoring agents and the communication overhead that arises from monitoring our case study. We have detected an average slowdown of the monitored service composition of just below 10% with a possible worst-case increase in execution times of up to 19%, according to the 95% confidence interval. The resource usage requirements of the monitoring agents are quite modest. We have measured a CPU usage of 3% to 4% and around 20MB of used heap space. The average bit rates for the exchange of measurements in the local domain were 3.06Kbit/s and for communication across administrative domain boundaries they were 0.48Kbit/s. Taken together with our analytical model this confirms small communication overheads and good scalability. However, in its current implementation, our approach might not scale well to massive deployments with thousands of nodes.

The performance cost from monitoring a service composition to support data-driven diagnosis is not prohibitively expensive. This is particularly true, if we consider that our prototype does not employ any performance optimisations and leaves some room for improvement in that area. Even in its current state,

the overheads are tolerable by modern servers and networks. Whether or not the measured slowdown of a service composition is at an acceptable level depends to a large extent on the savings in downtime that data-driven diagnosis can help system operators to achieve. A precise answer to this question is only possible on a case-by-case basis. However, as we have explained at the beginning of this thesis, an increase in the number of successful diagnoses and a reduction of their diagnosis times will serve to realise such savings. The overheads are not negligible and therefore the application of data-driven diagnosis needs to be justified by its benefits, but as far as our first research question is concerned, we can confirm the feasibility of providing support for data-driven diagnosis of middleware-based, cross-domain service compositions.

The second objective we addressed was to quantify the effects of data-driven diagnosis in order to determine its benefits. We measured the effect in terms of the success rate and the diagnosis times it can enable system operators to achieve. We have compared the success rates of participants in the effect group and control group. Participants using data-driven diagnosis handled the injected failures with a near-perfect success rate. Control group participants, on the other hand, achieved rates of $\frac{2}{3}$ to $\frac{4}{5}$ at best.

These results confirm our expectation about the effect of data-driven diagnosis on the number of successful diagnoses. Data-driven diagnosis substantially enhances the ability of a system operator to diagnose more failures correctly within the ten minute time limit we have set. What this means in practice is that a system operator using our approach is able to conclude more failure diagnoses successfully within a given amount of time than someone using something akin to the currently prevalent approach. This represents an important advantage of data-driven diagnosis.

In our examination of diagnosis times, we have found a statistically significant reduction of 22% on average across all six failures. These savings increase to 55% for failures whose cause lies in another administrative domain. We were unable to confirm a statistically significant reduction of diagnosis times when data-driven diagnosis was applied exclusively to failures of the local domain. However, on closer inspection of the times observed for individual failures, we found that data-driven diagnosis has contributed to diagnosis time savings across all failures except one. The higher than usual diagnosis times observed for Monere participants for this failure were due to a deficiency of the dependency graph. Overall, we can confirm our expectation that data-driven diagnosis enables system operators to achieve significantly shorter diagnosis times.

Before discussing the wider implications of these results, we briefly review our insights about the utility of dependency graphs. We argued that dependency graphs could effectively assist system operators by providing them with an overview of how the many parts of a system depend on each other. During the experiment we were able to observe how these graphs guide system operators and help them to focus the diagnosis process. For example, participants would often start by determining at what point of its execution the service composition had failed or stopped making progress and then proceed by examining the dependencies of the relevant component (e.g., a Web service, BPEL process). The ability to do so is particularly useful in the absence of any useful error messages. Another example is the ability to spot components that are common dependencies of several misbehaving components and that sug-

gest themselves for closer inspection. The dependency graph also facilitates the identification of remote components that could potentially impact the local deployment.

However, our experiment also revealed the negative effects that are the result of missing information in a dependency graph. As already reported, Failure 5, which is the exhaustion of disk space that eventually leads to the unavailability of the Condor scheduler, was particularly difficult for Monere participants. Two thirds of the failed diagnoses in this group were due to this failure and we have observed significantly higher diagnosis times compared to the Control group. We were able to identify the lack of data dependencies in our dependency graphs as the common cause of this. This observations suggests that missing relationships in a dependency graph can be so misleading during diagnosis as to reduce an operator's performance to below that of someone without access to such information about system structure. Whatever approach is used to obtain dependencies, it must lead to accurate and complete graphs as system operators tend to act as if they were.

These insights enable us to reason about the circumstances under which the application of data-driven diagnosis is justified. In order for service providers to decide whether to employ data-driven diagnosis and thereby bear the associated performance overheads, they need to consider two attributes of their service offering. First, one needs to have an accurate idea of the downtime cost per unit of time. It is important to know how much cost is incurred, if a particular application that has been implemented as a service composition becomes unavailable or otherwise non-performant. 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 [Skene et al., 2010]. 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 such information should become available, from detailed measurement-based failure studies on large service compositions.

In general, we can conclude that the application of data-driven diagnosis is justified when it is applied to service compositions 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. In addition, data-driven diagnosis is clearly of benefit for service compositions that integrate numerous resources across administrative domain boundaries and where failures can propagate their effects across these boundaries.

There are a number of papers that have made intuitively appealing arguments about the effects of monitoring and for its potential benefits on the availability of software systems. Consequently, the research questions that we have addressed were inspired by a number of other papers.

In [Plattner, 1984] the author argues that online monitoring should be of considerable benefit for the debugging of distributed systems and outlines a set of basic requirements. [Katchabaw et al., 1997] gives reasons for the importance and potential benefits of instrumenting distributed systems in order to facilitate their management. In [Vogels, 1996] Werner Vogels explains how failure reporting and the task of identifying failure causes could be improved by exploiting a wide array of readily available data about

system operation. In [Vogels and Re, 2003] he argues that such monitoring may be useful for porting process group semantics to Web services. Finally, [Chandra et al., 2008] berates the lack of monitoring approaches that primarily focus on the ability to support failure analysis in large systems. The ideas, arguments and open questions in these papers have led us to investigate and quantify the utility of making a wide array of monitoring data available to system operators for the diagnosis of system-level failures.

We build on their work in three ways. First, we have developed what we refer to as a data-driven approach based on extracting information from a middleware-based system in the form of various metrics and dependency graphs. Second, we performed the first evidence-based determination of the effects that making such data about system operation and structure available to system operators has on the task of failure diagnosis. Finally, an additional aspect of our work is that we argue that the provision of dependency graphs is an important feature to support system operators to perform failure diagnosis of failures in large, cross-domain systems in a more efficient manner.

We have investigated the use of data-driven diagnosis for the early root cause localisation of system-level failures. We need to bear in mind that in some instances, it may be necessary to follow up such an initial determination of a failure cause with more specialised tools, before a repair can be effected (i.e. in order to identify exactly how a deadlock came about). Data-driven diagnosis does not cater for such detailed diagnoses. In such cases and where the above criteria are not met, the net benefits of data-driven diagnosis will be less extensive, even though service providers should nevertheless experience some increase in success rates and a decrease in diagnosis times.

This concludes our investigation of the benfits data-driven diagnosis can provide under various conditions. Next, we turn our attention to the automation of failure detection.

Chapter 6

Classifying System Operation

In this chapter we motivate the final two research questions that this thesis addresses and provide the necessary background in machine learning to understand our work. The first remaining research question is concerned with the accuracy with which a particular type of machine learning algorithm (one-class classification) can learn to detect system-level failures based on multi-variate monitoring data about system operation. After outlining the motivation for this automated approach to failure detection, we introduce the reader to classification-based machine learning, before presenting the details of the particular algorithm we apply to this problem along with the rationale for its selection. The second research question addresses how to select subsets of metrics in order to learn highly accurate classifiers for failure detection. We motivate this question in the penultimate section where we discuss, among other issues that arise from the application of our approach in practice, the challenges that result from trying to discern structure within the large space represented by multi-variate monitoring data. We close this chapter with a review of existing approaches that apply machine learning and statistical modelling to the detection of failure conditions in distributed software systems.

6.1 Motivation

The mean time to repair a failure has three constituents. As we have briefly explained in Chapter 1, $MTTR = MTT_{detect} + MTT_{diagnose} + MTT_{repair}$. In many cases, the performance of appropriate repair actions is based on successful diagnosis that identifies the cause of a failure. Diagnosis begins only once a failure has been detected in some way. So far in this thesis, we have relied on end users to notify system operators of any extraordinary behaviour that an application exhibits. Such an approach is far from optimal. Instead, we want to detect the presence of a system-level failure as quickly as possible and whenever possible we want to do so before end users encounter and complain about a problem. Faster failure detection can enable system operators to begin the process of diagnosis, and accordingly to apply corrective action, sooner. In order to further reduce the overall time to repair a failure, detection needs to work in machine rather than human time scales.

Manual monitoring for failures is problematic for a number of reasons apart from resulting in relatively long detection times. One major drawback is that it necessitates continuous monitoring by a system operator. This involves the observation of a large number of components, most of which vary

in their failure modes. An operator needs to know what aspects of a component's operation can reveal the presence of certain failures or incorrect operation. Furthermore, there may be long periods of time during which no events of interest occur, which makes it difficult to maintain a high level of attentiveness and concentration. Another issue is that the error reporting facilities of many middleware-based service compositions are often not adequate for effective failure detection. For example, failed invocations often result in HTTP error codes, which are sometimes recorded in a log file. This may be an appropriate way to report failures in an interactive system in which a user is immediately presented with the error message and can possibly react to it based on the nature of the request that has failed. However, in an automated system, in which an initial request sets off a chain reaction of activity among several subsystems and partner services, human beings may only check for progress periodically. Third, even a mechanism that can capture a variety of events that indicate failures and raises alerts accordingly, would not be able to detect failures whose occurrence is less obvious and does not result in an error message. For example, in the Propaganda example application, the retrieval of relevant news messages may be slow and incomplete due to overload at the news services supplying them. But while the slowdown of some parts of the service composition is likely to be regarded as faulty behaviour of the service by end users, it is usually not associated with error messages. Fourth, in cross-domain systems, a system operator may not be privy to the occurrence of a failure that has occurred in a remote administrative domain other than through its eventual effects on part of the local infrastructure. And finally, a failure detection process that is largely manual incurs substantial labour costs. These factors combine to make manual monitoring for failures time-consuming, error-prone and expensive.

We want to automate the detection of system-level failures in middleware-based service compositions. A suitable detection mechanism should require only minimal human intervention in order to work accurately. We avoid semi-manual approaches which rely on the definition of knowledge bases containing rules or other representations of information on how to detect failures, because it is difficult to ensure that such knowledge bases are both accurate and comprehensive. The provision of such information requires intimate familiarity with middleware-based systems. Furthermore, there remains a risk that a knowledge base of rules for the detection of failures always lags behind the latest unseen failure.

Instead, we propose the automation of failure detection by applying outlier detection algorithms to monitoring data. Our monitoring approach provides live measurements at runtime and makes this available alongside historic records of multiple metrics. This forms a repository of multi-variate data about the behaviour of many of the components comprising a given service composition and contains many of the attributes that system operators are likely to take into consideration in order to detect failures. This stream of monitoring data captures both normal system behaviour as well as faulty operation and could therefore be useful for outlier detection to automate the detection of failures.

The novelty of the work on automated failure detection which we describe in this and the next chapter is comprised of three elements. First, we evaluate the utility of a one-class classification algorithm that has not been previously applied to the problem domain of detecting failures in distributed software systems based on monitoring data about their operation. Second, we apply a large set of general metrics

about system operation to this classification-based mechanism in order to train it to discriminate between normal and faulty operation. As we will discuss shortly, many related approaches usually rely on small sets of metrics that are often hand-picked in order to maximise performance. And third, in addition to approaches that do use a large number of metrics, we evaluate several strategies to select subsets of metrics, which make learning an accurate classifier more tractable.

The next chapter presents our evaluation of this approach to automate failure detection and sheds light on two research questions. First, we find out whether a high level of detection accuracy is possible when applying one-class classification to multi-variate monitoring data. Second, in order to make it tractable to learn a model that discriminates between normal and failure operation, it is necessary to reduce the number of attributes in the dataset[1]. We examine several strategies for choosing subsets of metrics from an otherwise large dataset and learn how to perform such a selection in a way that achieves high levels of detection accuracy.

6.2 Classification

Statistical and machine learning approaches can discover structure and patterns in large datasets that would otherwise remain hidden [Witten and Frank, 2005]. The learned models can then be used to make predictions about future values or to group data items into clusters of similarity. There are numerous approaches that merit an investigation of their utility for the automation of failure detection and we will point to references to some of these during the course of this chapter. We focus on classification-based machine learning and outline why the application of one-class classification to multi-variate monitoring data offers itself as a particularly promising approach to automate failure detection. In short, one-class classification is an outlier detection technique that is applicable to situations where there is more data about a target class (e.g., normal system operation) than about outliers (e.g., failures). The particular algorithm that we use has so far not been evaluated for the automation of failure detection in distributed software systems.

Classification-based learning is a form of inductive learning, in which a model is learned based on a set of examples presented during training. In machine learning, an example is also often referred to as an *instance*. An instance is characterised by a set of attributes and their values. The goal of a classification-based learner is to learn a model that generalises from the set of training instances to yet unseen ones. Provided with a new instance as input, the learned model should classify the instance as belonging to one of several classes. The training instances are fully labelled. That is, their class labels are known and made available to the learner. In order to illustrate this with an example, consider that we are dealing with instances of cups of different kinds of tea. Some of the attributes that are relevant in order to characterise a cup of tea might be its temperature, its opacity, its colour and the type of tea. The first two attributes are multi-valued and the last two are nominal ones. We could designate the type attribute as the class attribute. During training, the learner would be presented with fully labelled examples that reveal their correct classification. After training, its goal would be to classify new cups of tea that it has not encountered before (i.e. this cup of tea belongs to the class of “peppermint teas”). The

[1] We discuss this issue in more detail in Section 6.4.

task performed by a classification-based learner is to classify instances of an object into distinct classes according to their characteristics.

A decision tree learner is an instance of a classification-based approach whose classifier is represented in the form of a so-called decision tree. The nodes of a decision tree represent tests on the values of an attribute. The edges partition the instance space based on the particular values for these attributes. The leaf nodes represent the classes an instance can belong to. A new instance reaches one leaf node after traversing through the attribute value tests defined by the tree.

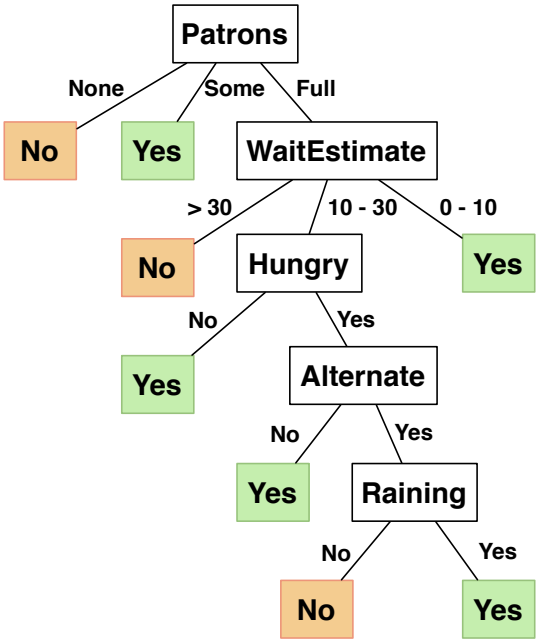


Figure 6.1: An example decision tree about whether or not to wait for a table in a restaurant.

An example decision tree is shown in Figure 6.1, which we have adapted from Chapter 18 of [Russell and Norvig, 1995]. The tree classifies a set of current circumstances into a decision about whether or not we should wait for a table to become available at a restaurant. The root node (*Patrons*) tests how many patrons are at the restaurant. Patrons has three outgoing edges, two of which result in a "yes" or "no" classification and a third one, $Patrons = Full$, that leads to another test represented by the node *WaitEstimate*. This node tests the current situation for its estimated waiting time for a table. If the waiting time is more than 30 minutes or less than 10 minutes, then the resulting classification is "no" or "yes" respectively. Otherwise, we move one level further down in the tree to evaluate the situation based on whether or not we are hungry. This process continues until the current situation has reached one of the leaf nodes.

Widely used algorithms for decision tree learning, such as C4.5 [Quinlan, 1993], construct decision trees in a divide-and-conquer manner. The goal is to build a tree that has as few tests and hence as few levels as possible. At each new level of the tree, the algorithm chooses an attribute that separates the highest number of examples into distinct classes and then it proceeds with the remaining attributes

until all training examples have been assigned to one of the leaf nodes. Attributes that can classify more examples into distinct classes are preferred at each step. The best attribute is selected based on concepts from information theory, such as information gain and gain ratio, which we do not cover here.

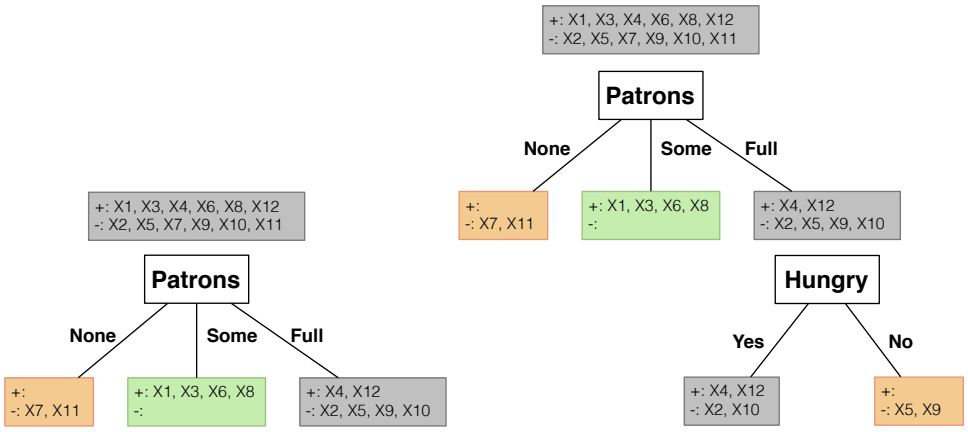


Figure 6.2: Two example steps of a decision tree learner that show how tests on attributes separate positive and negative examples into distinct classifications.

Consider the following example of learning a decision tree, which we have again adapted from Chapter 18 of [Russell and Norvig, 1995] and which is illustrated in Figure 6.2. $X_1, \ldots, X_{12}$ describe training instances, each of which specifies values for several attributes, such as *Patrons*, *Hungry*, *Wait-estimate* and a correct classification that indicates whether or not we should wait given these values. Figure 6.2(a) shows how the three possible values of *Patrons* splits the examples. The split results in six of the examples being assigned to either a negative or a positive leaf node, while $Patrons = Full$ leaves six more examples unclassified. In the next step of the learning process, shown in Figure 6.2(b), the remaining examples are tested against the possible values of the *Hungry* attribute. This test classifies two of the remaining training examples in a new negative leaf node and leaves four examples unclassified. This algorithm continues in this manner until all examples have been fully classified. The proportion of training instances at the individual leaf nodes represent class probability estimates.

In general, we distinguish between *supervised*, *unsupervised* and *semi-supervised* learning approaches. A supervised learner is given a set of fully labelled training samples. The training set consists of a number of instances, each of which have the same attributes including a class attribute. The learner can then construct a classifier that maps instances to their correct class. If all data is unlabelled, unsupervised approaches, such as clustering algorithms, group available data into groups based on some notion of distance given their attribute values. Finally, semi-supervised approaches are applied in situations or in problem domains where fully labelled data is not available, but in which it is known that the available training data belongs to a single class (i.e. normal data). Here, the learner needs to construct a model of only that one class in order to discriminate it from all other classes.

The decision tree algorithm is an example of a supervised approach as it is trained on fully labelled

data. Decision trees can represent Boolean functions. As such they classify instances into one of two classes, which is also known as *two-class classification*. This would appear to represent an approach that is suitable for distinguishing between monitoring data that describes normal or abnormal operation. However, the accuracy of a supervised two-class classifier depends on being trained on a large number of instances for each of the two classes. If one of the classes is only represented by a relatively small number of training instances, the learned classification model is likely to be inaccurate. Furthermore, a two-class classifier that is trained on only one class does not really discriminate between target and non-target patterns. Instead, it simply fails to classify instances that are unlike the classes encountered during training.

There is a limited number of ways in which we can break a service composition in order to observe failures and collect training data for a classification learner. While it is possible to apply failure injection in order to provoke a wide variety of failures under different conditions, it is impossible to know which additional failures have not been elicited or encountered in this manner. Furthermore, in a cross-domain system, such failure injection experiments require the cooperation of several administrative domains. However, observation of a service composition under normal behaviour is relatively inexpensive and can be achieved by using a monitoring framework such as ours. In our problem domain it is easier to obtain a large number of observations of normal system operation than it is to observe a large number of failures. Therefore, a semi-supervised approach that learns a model of normal behaviour and can then detect significant deviations from this model seems to be a more promising approach for our purposes.

6.3 One-Class Classification

We want our automated failure detection mechanism to be able to separate normal from abnormal monitoring data after having been trained only on data about normal system operation. Anomaly detection techniques offer themselves for the implementation of such a mechanism. There are numerous techniques [Chandola et al., 2009] and one of them is one-class classification [Moya et al., 1993], [Moya and Hush, 1996]. A one-class classifier is trained on data that represents what is known as the *target class* without requiring exposure to non-target class data. The classifier surrounds the target class space with a boundary in the form of a hypersphere of as many dimensions are there are attributes per instance. This hypersphere separates the instance space into a target and a non-target class region. Instances whose attribute values place it outside the hypersphere are considered to be outliers. In our context, the target class corresponds to data about normal operation and outliers represent monitoring data that is indicative of a system-level failure. In principle, one-class classification should be able to distinguish between normal and abnormal monitoring data based solely on target class training data.

Figure 6.3 shows an example of a hypersphere defined by a one-class classifier. The objects are depictions of trees and the target class consists of different types of natural trees, such as birch and pine. The outliers in the example are graph theoretical representations of a tree. In order to keep the illustration simple, the instances in this example are, unrealistically, characterised by only two attributes. Trees of different types can all belong to the target class. The common defining characteristic of members of the

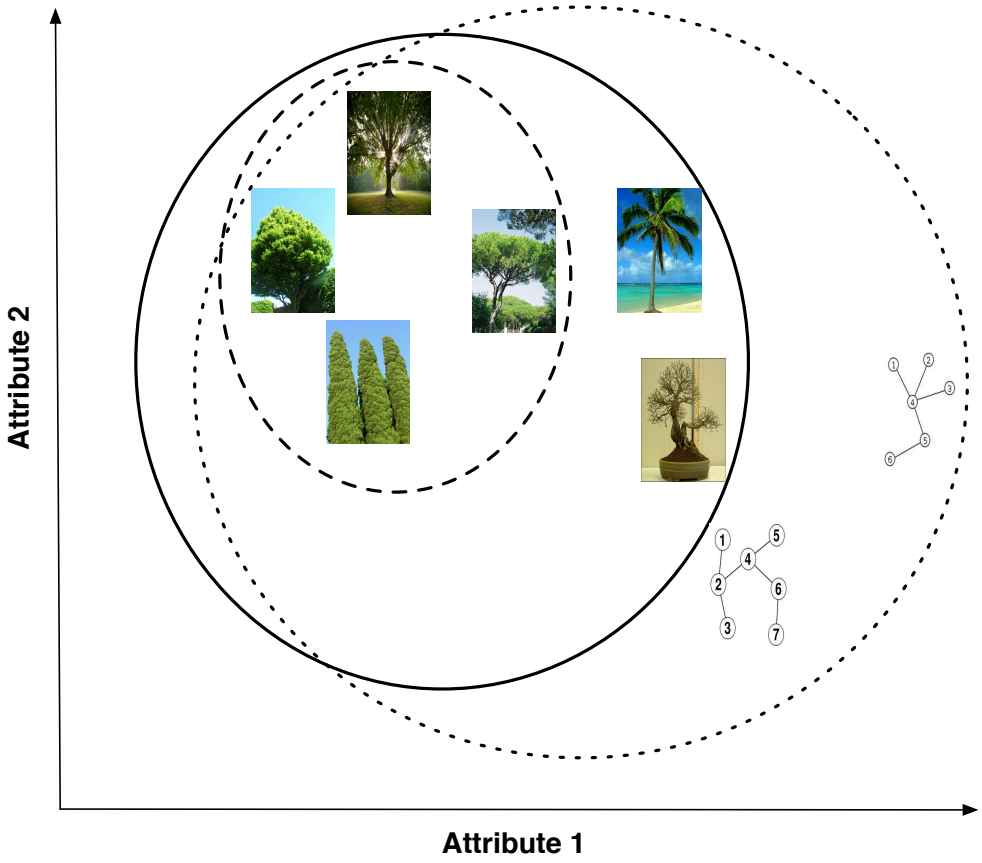


Figure 6.3: Two-dimensional hypersphere that separates depictions of natural trees from graph-theoretic representations. Different hyperspheres result in varying rates of false negatives and false positives.

target class is that they are trees as they occur in nature. This is important as data about normal system operation may be observed under varying workloads, involve the invocation of different operations and vary in other ways. If the set of target class instances formed a highly homogeneous set, the problem of assigning membership and of detecting outliers would be much simpler. Instead, the goal of the one-class classifier is to define a well-fitting boundary around the target class that will take account of this variability of what is considered normal without admitting non-target class instances. If the resulting hypersphere is too tightly defined around the target class, the probability of mistakenly classifying target class instances as outliers increases. For failure detection, this leads to a higher rate of false positives. That is, normal instances are incorrectly classified as outliers or failures. Similarly, as the region defined by the hypersphere widens, the classifier may miss a larger proportion of outliers. In our application domain, this would lead to an increased rate of false negatives or missed failures.

One common approach to one-class classification is to apply density estimators to the target class training data [Tax, 2001]. Density estimation results in a density curve that represents the shape of the data and models the probability distribution of a continuous random variable (e.g., a single attribute in the uni-variate case). The entire area under the curve is always equal to 1. The area under the

curve and between a range of values gives the probability for the occurrence of this range of values. This makes it possible to determine what area under the curve is occupied by the attribute values of a particular instance. Instances whose attribute values result in only small areas under the density curve, consequently have a small probability of belonging to the target class given the evidence provided by the training data. Such instances are considered to be outliers.

In our experiments in the following chapter we use a recent one-class classification algorithm that has been proposed in [Hempstalk et al., 2008]. It differs from previous algorithms in that it uses both density estimation and two-class classification in order to define a more accurate hypersphere. The algorithm combines the probability density function that results from density estimation of the target class data with class probability estimates obtained by applying a two-class classification algorithm to artificial data that represents an outlier class. The authors are able to demonstrate that their approach leads to better performance than using either component mechanism on its own. This result holds, if there is not a large number of non-target class data available for training. In cases where such data is available, a two-class classifier may outperform the one-class classifier [Hempstalk and Frank, 2008].

We provide a basic overview of the operation of this algorithm here and refer the interested reader to [Hempstalk et al., 2008] for a more detailed and mathematically accurate exposition. As its first step, the algorithm estimates a density function from the target class training data. It then selects regions of values under the curve with only small probability of belonging to the target class given the shape of the training data. Based on this, the algorithm can create artificial data that represents a plausible second or outlier class. By using regions of low probability based on the density function, the algorithm can ensure that the generated data items are as close as possible to the target class. This allows for training an accurate two-class classifier that discriminates between target class and artificial outlier data without requiring massive amounts of artificial data. The learned two-class classifier provides class probability estimates that the algorithm combines with the estimated density function of the target class data. It then uses the combined model to calculate membership scores that indicate whether a given data item belongs to the target class. A detailed mathematical model of these procedures is given in Section 3 of [Hempstalk et al., 2008].

One-class classification represents a promising approach to automate the detection of system-level failures based on monitoring data about system operation. The algorithm we have chosen for this purpose has been shown [Hempstalk et al., 2008] to learn relatively accurate hyperspheres for a number of datasets in cases where non-target class training data is limited, as is the case in our application domain. One remaining question is how accurately it can predict system-level failures based on instrumented system state.

6.4 Practical Considerations

The evaluation of automated failure detection through one-class classification in the next chapter is based on a set of offline experiments that we perform on recorded monitoring data. In order to illustrate that this is in fact a practical approach, we briefly discuss three issues that affect the application of such a mechanism in practice. In particular, we consider its integration into the existing monitoring

infrastructure. We compare the characteristics of our monitoring data to the training requirements of the classification algorithm. And finally, we explain how the large space represented by this monitoring data impairs the algorithm's ability to learn an accurate classifier and how we are going to address this issue.

The one-class classification algorithm could be integrated in the form of an extension to the Monere prototype monitoring framework. After a new service composition has been deployed and once a sufficient amount of data about its operation has been collected, a system operator can train the failure detection module on this data. After the training phase, the module can process incoming monitoring data at runtime and raise an alarm as soon as it detects system state that it considers to be abnormal. The alert could be communicated through a web-based front-end, such as the one system operators use for diagnosis of failures. We do not address how the classifier could adapt its model dynamically in order to improve its accuracy based on new observations that have become available since training was completed. Instead, we assume that a system operator will determine if and when the classifier needs to be adapted and re-train it.

Most learning algorithms impose certain requirements on the kinds of data used during training and subsequent processing. Monitoring data in a format that is suitable for a human consumer may not be equally adequate for learning algorithms. The Monere prototype continuously collects measurements of a variety of metrics from the discovered components at the collection intervals that have been defined for them. This results in a repository of timestamped measurements, which together form a multi-variate time-series. Classification algorithms do not usually handle data in the form of time-series. Instead, they work on examples or on instances of the objects that are to be classified. In order to serve as input to the one-class classification algorithm, the raw monitoring data needs to be converted into instance data.

For this purpose, we have built a simple tool that divides the measurements into ten-second windows and calculates aggregates over these windows for each continuous metric. Based on the monitoring data for host *huey* each instance, which now represents a ten-second window of system operation, contains 515 attributes. The number of attributes is higher than the number of metrics monitored on the host as some metrics are represented by several attributes. For example, the availability metric of a component has a Boolean attribute to denote the current status, but also results in attributes that represent related statistics, such as the time between failures and the average time for a component to become available again. The one-class classifier requires instances to be labelled as normal or outlier instances. We have added an additional class attribute in order to record this information. Normal and outlier instances have been combined into a single large data set and the one-class classification algorithm simply ignores all outlier instances during training.

The choice of the time window size warrants some discussion. In [Wang et al., 2011] the authors discuss online anomaly detection for data center management and mention the difficulty of determining an appropriate historical period to use for aggregation. They suggest that it might be feasible to address this issue by running an algorithm on data that is aggregated over several window sizes in parallel. However, for the set of metrics and collection intervals in our case study, ten-second intervals represent a good trade-off between several factors. The collection intervals range from 1 second to ten minutes and

of the small collection intervals, ten seconds is the most frequently used one for numeric metrics. We want to generate a large number of instances from our monitoring data in order to be able to successfully train a classifier and confront it with challenging test sets. At smaller intervals (e.g., 1-second and 5-second), we are able to generate a greater number of instances, but there will not be many new measurements from one instance to the next and many of the generated instances would be duplicates that do not provide much additional value for training and testing. Furthermore, we want to detect failures as quickly as possible and avoid missing any. As we consider relatively long-lived failures, it is safe to assume that a failure that is present in any 5-second interval will also be present in 10-second intervals. Going from 5-second to 10-second intervals does not introduce a considerable delay and does not risk losing any valuable information. However, the same may not be true at larger intervals, where data that is indicative of a failure may become less prominent due to a larger amount of normal data that may fall within the larger interval. In addition, larger time windows result in a smaller number of instances. For our case study, ten-second intervals represent a good compromise between creating a large number of instances from the monitoring data, which is useful for training and testing of the classifier, without producing too many duplicate instances while at the same time still being able to quickly detect failures without risking missing any of them.

The final issue that we consider here concerns the ability of the algorithm to learn an accurate classifier within the large space that multi-variate monitoring data represents. This space, which the classifier separates with a suitable hypersphere, is known as the *feature space*. The feature space is defined by the instance attributes and their possible values. Our monitoring data consists of a large number of metrics. Each metric of a component is represented by an instance attribute. Most of the metrics that comprise our monitoring data and hence their corresponding instance attributes have continuous values. This creates a large and highly-dimensional feature space. The number of attributes of the instances dictates the dimensionality of the feature space; the number of possible values along each dimension influences its volume. Each instance, which represents a window of monitoring data as we will explain in the next chapter, occupies a particular location in the feature space as determined by the values of its attributes.

The one-class classification algorithm learns a model of normal instances by examples from a necessarily finite number of training samples. At a fixed training set size, the accuracy of the learned classifier (i.e. how well the boundary defined by the hypersphere fits the target data) decreases with increasing dimensionality of the feature space. As its dimensionality increases, a significantly greater amount of training data is needed in order to ensure that the learner can cover a sufficiently large part of the feature space and encounter as many different combinations of values as possible. This phenomenon is known as the Hughes Effect [Hughes, 1968].

It is necessary to somehow fragment the feature space in order to enable the algorithm to learn an accurate classifier. We deal with this issue by examining several strategies for choosing subsets of metrics in a way that leads to highly accurate classifiers as we explain in Chapter 7.

6.5 Related Work

We review related work that performs failure detection by analysing some form of information about the operation of a software system. We divide approaches into model- and rule-based approaches and ones that apply statistical and machine learning techniques to some element of system state. Some of these approaches can also afford root cause localisation of failures.

6.5.1 Model- and Rule-based Approaches

Model- and rule-based approaches are concerned with how to correlate a large number of system events or alarms in order to detect the occurrence of failures. Network management systems can produce a large number of alarms when problems are experienced by the network infrastructure. It can be challenging for a human system operator to filter the information efficiently. In particular, individual alarms may not provide much useful information that leads to the detection of a failure. However, several alarms that occur shortly of each other may provide useful clues. In [Gardner and Harle, 1998] the authors present a correlation language and a simple architecture to process a large number of alarms based on the rules expressed in this language. A domain expert is expected to write correlation rules that capture how several alarms can be representative of certain types of failures. For example, if two alarms of a certain type occur within a specific time limit from each other and are then followed by a third event of a third type within another time limit, then this may indicate that a particular failure has occurred. This approach depends on a domain expert being able to enumerate most of the ways in which a set of events can be correlated with the occurrence of different kinds of failures in a given system.

A similar approach is presented in [Klinger et al., 1995] for the detection of failures in complex IT systems. A domain expert encodes model knowledge that then enables a management component to correlate observed events with different types of failures. The encoded system model includes the managed objects and their relationships with each other, a list of failures, a list of symptoms for each failure in terms of the variables in a management information base and a set of observable events. The approach uses this description of failures to assign codes to the events that are generated by a particular failure. It finds the minimum set of events (or the smallest code) that is sufficient to detect and distinguish each type of failure in the model. Relying on such optimal subsets of events reduces the amount of event processing that the approach has to perform.

In [Yan et al., 2005] the authors propose to diagnose failures in cross-domain Web service compositions by using model-based diagnosis. Model-based diagnosis reasons about a model of a system, its components, interactions and the properties of these interactions in order to determine the set of components that can have led to the occurrence of a particular failure. The authors propose to convert the information contained in the specification of a BPEL process into a Discrete Event System (DES) model and describe a mapping from the BPEL language constructs to a DES. A DES is composed of a set of states, events, transitions between these states as well as an initial and a final state. It is possible to determine all paths of events and system states that occur during the execution of a BPEL service composition as this information can be obtained from a BPEL engine. This makes it possible to reason about the component that may be responsible for a particular fault. Detection of a fault is done simply

by noting that a particular application-level fault has been thrown in the BPEL process. Diagnosis then proceeds to infer the likely cause. For example, if an activity in a BPEL process has thrown a fault, then it can either be due to a fault in the activity itself or in any of the services that generate or change part of the input of this activity. In this manner, the likely BPEL activites and Web services that are responsible for a fault can be narrowed down. This approach does not deal with fault detection, but demonstrates how the data provided by a BPEL engine can be used to diagnose the cause of application-level faults at a high level.

Another approach to determine the likely causes of application-level faults in Web service compositions that relies on model-based diagnosis is proposed in [Ardissono et al., 2006]. The model consists of a precise specification of the control and data flow of each Web service in the composition. This includes a list of its operations, the messages it exchanges, information about dependencies between operation parameters and the fault messages that can be generated. A local diagnoser component is associated with each Web service in the composition. When a fault is thrown in its Web service, the local diagnoser analyses all messages that the service has exchanged with its peers and examines the parameters used in the operation invocations. This results in local hypotheses about the set of Web services whose behaviour may have given rise to an observed fault. The hypotheses are submitted to a global diagnoser component, which may then invoke other local diagnosers. In the end, if there only remains a single hypothesis about which Web service is likely to be responsible for the fault, then a suitable recovery strategy can be executed. Otherwise, the approach would notify a human operator who is presented with the list of suspected Web services.

Our approach to failure detection differs from the semi-manual approaches discussed here as it does not rely on the provision of expert knowledge about the composition of a system, its failures or their symptoms. Such an approach might be difficult to realise in our context in light of the large number of different components that are involved in supporting the execution of a service composition. Model-based approaches that operate on BPEL workflows address the diagnosis of application-level failures whose detection is adequately provided for by existing mechanisms. Our approach to failure detection considers failures that cannot, in most instances, be adequately handled by an application and that can arise in any of the layers that comprise a service composition.

6.5.2 Statistical and Machine Learning Approaches

There are numerous approaches that process the information contained in log files in order to detect failures. In [Lim et al., 2008] the authors discuss a few statistical techniques that they have applied to large log files that were produced by an enterprise telephony system. It is known which log entries correspond to crash failures. The authors describe a simple pre-processing step to deal with the large number of original log entries (11 million). They extract the process names and payload information of each log entry and remove duplicates to obtain 2.5 million unique log messages. They further reduce this down to $11,000$ by clustering these messages based on the similarity of their content. They then describe three approaches to detect failures. The first one identifies log entries that repeat themselves around the time that a failure is known to occur. In the second approach changes in the overall message

frequency are observed. The intuition behind this approach is that processes tend to log considerably more messages soon before a failure occurs. The third approach analyses the frequency of individual message types. The idea is to detect message types whose frequency exceeds a certain threshold around the time that a failure occurs. All three analyses lead to information that can be used to detect failures.

The work in [Carrozza et al., 2008] applies document classification techniques to log files in order to detect and localise transient failures in software systems that have been built out of off-the-shelf components. The available log files contain information about the severity of logged messages, timestamps, the IP address of the source component that logged the message, process and thread identifiers, the amount of used memory at the time and the payload of the message. Detection works by first training a one-class Support Vector Machine on examples of normal log messages so that the resulting classifier detects outliers as potential failures. Each component has its own local detector and if its classifier suspects a log entry to be representative of a failure, the local detector creates a document that contains the relevant log entries. The documents from all diagnosis components that have recently fired alarms are merged into a so-called Alarm Document (AD). This forms the basis for localisation. Relevant features are extracted from the document and a second classifier analyses the frequency with which each word appears in the collated log messages. The result is an array of probability values that indicate which of n fault classes is the most likely cause of the detected failure. The list of relevant fault classes can be contributed by a domain expert. The approach combines local, per-component detection of failures with a global analysis to localise their cause.

We have already discussed the Chopstix [Bhatia et al., 2008] monitoring approach in Section 3.6. The authors also describe how Chopstix can be applied to the detection of certain failure conditions. It uses simple heuristics applied to the monitoring data to detect failures. For example, high CPU utilisation and a low L2-cache-miss rate is regarded as a symptom of a busy loop, which uses a great deal of CPU without producing any results. Similarly, an increase in user memory that becomes allocated over time is an indicator for the presence of a memory leak. Chopstix relies on expert users to encode such heuristics over the metrics it makes available.

In [Vilalta et al., 2002] the authors provide an overview of different statistical and machine learning approaches that can be applied to several problems as they arise in systems management. One of these is the short-term prediction of abnormal behaviour. The goal is to be able to predict threshold violations of the performance of a web server. The metric the authors select for this purpose is the number of HTTP operations that the server performs per second. They explain how to use change-point detection based on the Generalized Likelihood algorithm to detect significant changes in the metric values over time. The metric values are aggregated over a time window that starts at the last detected change point and continues to the present. This value is compared to a window that stretches some time into the past, so as to provide a sufficient number of observations. In this example, the authors use a hand-picked metric that is known to be useful for failure detection.

The use of Tree-Augmented Naïve Bayesian Networks (TAN) to determine correlations between system-level metrics and the violation of service-level objectives (performance objectives) in a standard

three-tier web application is evaluated in [Cohen et al., 2004]. TANs are efficient Bayesian networks that have been successfully applied in problem domains such as financial modelling and medical diagnosis. Their results are interpretable, which can support localisation of failure causes. A total of 124 metrics about the operation of a web application are collected. These include metrics, such as the user CPU time, the number of physical disk reads, the variance in swap space allocated to the database server and so on. As the number of metrics is too large in order to enable a TAN to learn an accurate classifier, the authors present an algorithm to select subsets of metrics. This algorithm adds one metric at a time to the subset. At each step, it selects the metric that results in the maximum improvement in accuracy out of all remaining metrics. The results of the experimental evaluation of the use of TANs reveals a detection rate of about 90% at a false alarm rate of approximately 6%. Furthermore, the authors find that small numbers of metrics (three to eight metrics) are sufficient to accurately predict SLO violations and that combinations of metrics perform better than individual ones.

The approach in [Guo et al., 2006] models correlations between pairs of metrics to achieve fault detection in multi-tier Internet services. The approach can detect conditions such as the presence of busy loops, memory leaks and deadlocks. The metrics used are HTTP requests as they are observed in server logs, CPU usage, memory usage and the number of threads and EJBs. The approach taken is to create a model of the normal behaviour of the service and to then detect significant deviations from that. Correlations between pairs of metrics are quantified using Gaussian Mixture models. In order to select suitable pairs of metrics, any combination of two metrics is evaluated by modelling its correlation and then testing it on new data points. Over time, a fitness score is calculated for each pair and the ones that best fit the real data distribution are selected for use at runtime. If the correlation between a pair of metrics weakens at runtime or becomes broken, then this is seen as evidence that an anomaly involving these two metrics has arisen. The authors are able to demonstrate the feasibility of their approach by detecting anomalies between thread and CPU usage and between HTTP requests and memory usage.

Pinpoint [Kiciman and Fox, 2005] is a system for the detection of failures in Internet services that does not rely on a varied set of metrics about the operation of an application. Instead, Pinpoint captures component interactions and models the shapes of runtime paths of requests by tracing their traversal among the components of a Java application server. This requires its instrumentation to assign unique request identifiers to end-user requests and to propagate these among components. The runtime paths are assigned weights in accordance with the number of requests that have been observed between the components that they connect. As long as there is a large number of requests in the system, it is possible to detect significant deviations from the learned model by comparing it to more recent observations using a χ^2 test. The same instrumented state is used to learn path shapes of requests. Path shapes are represented as a kind of call tree whose nodes are the components involved in request processing and whose edges record the number of times that a component calls another one. This makes it possible to detect deviations by comparing the expected probability of transitions as given by the learned path shapes to the recently observed transitions. The authors report a failure detection rate of 89% to 100%.

Statistical modelling has also been applied to gain insight into software engineering processes.

In [Zimmermann et al., 2012] the authors set out to find out which factors are the most influential ones in affecting the reopening of bug reports. They use versions of a large commercial operating systems as a case study and perform a survey with a large number of stakeholders to obtain information about the various reasons that bugs get reopened. Based on the survey data, the authors then build three logistic regression models and compare these in order to determine the actual factors that appear to have the strongest influence on the likelihood of a bug being reopened. They establish a positive correlation for factors such as whether or not the person opening the bug was a temporary employee and whether the opener and the person assigned to fix the bug were working under the same manager. This is an example of applying quantitative models to gain insights into different aspects of software engineering processes (i.e. bug fixes and re-opening). Our efforts are more closely related to others that apply machine learning techniques to runtime data about a software system in order to classify its state as either faulty or normal on behalf of a system operator. We aim to automate the detection of failures at runtime.

There are several differences and similarities between our efforts and those we have reviewed here. The first notable difference is in the number and types of metrics used. Some approaches (e.g., [Kiciman and Fox, 2005]) develop custom metrics in the form of runtime paths and use this to model normal behaviour and to detect anomalies. Others (e.g., [Vilalta et al., 2002], [Guo et al., 2006]) rely on their expertise to select metrics that are known to correlate well with certain failures and thereby reduce the overall number of metrics that are considered a-priori. In our work, we explore how to learn to detect failures within a large, varied set of metrics that describe the behaviour of the plethora of components that support the execution of a service composition.

The work described in [Cohen et al., 2004] takes a similar approach in that the authors evaluate the use of 124 metrics and apply an algorithm to select subsets of metrics that support an accurate classifier. We are able to add to this an evaluation of several strategies for this kind of feature selection. Our most successful approach is similar to the one used by [Cohen et al., 2004]. They choose metrics that maximise the overall accuracy of the Bayesian network; we choose ones that maximise the accuracy of the internal classifier of the one-class algorithm that we evaluate.

A third difference is the mechanisms used to process monitoring data and perform failure detection. All reviewed approaches differ in the techniques they use. Our work is the first one to evaluate the suitability of the particular one-class classification algorithm that we have presented in Chapter 6 for this problem domain. [Carrozza et al., 2008] use a different one-class classification algorithm and evaluate its application to data that they obtain from log files. They are able to successfully build per-component classifiers in this way, which is something that our approach cannot. This difference may be due to our use of different (mostly numeric) metrics or may be a consequence of the algorithm used.

The reviewed papers differ, to some extent, in the accuracy measures used for the evaluation of the proposed mechanisms. These differences are not significant, but should nevertheless be reviewed briefly. [Kiciman and Fox, 2005] uses precision (proportion of retrieved instances that are relevant) and recall (proportion of relevant instances that are retrieved) to evaluate their failure detection. These measures are primarily used in the evaluation of document classification or search techniques. [Cohen et al., 2004]

uses a custom metric called Balanced Accuracy, which describes the accuracy of the classifier with a single value. We prefer to consider the TPR and FPR separately as this provides more insight into the behaviour of our failure detection mechanism in practice. Furthermore, as we have explained, the cost resulting from imperfections in these two metrics is not equal for the task of failure detection. The cost of inaccuracy is also likely to vary in an application-specific manner. Therefore, and also because these measures are commonly used in the form of sensitivity and specificity for the evaluation of diagnostic procedures, we prefer to use the TPR and FPR directly to evaluate the accuracy of failure detection.

Finally, our efforts differ in the types of failures that are being considered. Some of the reviewed approaches deal with the detection of threshold violations of performance metrics. The system-level failures that we consider are much less clear-cut and more arbitrary in their behaviour.

We are able to confirm some of the results reported by [Cohen et al., 2004]. We have also found that while individual metrics are not sufficient, small subsets of metrics lead to the highest accuracy. The failure detection rates of other approaches, where they are reported, appear to be similar to what we have observed. However, it is difficult to compare these results across the different types of failures and methods of evaluation.

Conclusions

In this chapter we have motivated our goal of automating the detection of system-level failures by applying classification-based machine learning to the multi-variate monitoring data our approach makes available about the operation of a service composition. We have provided some background knowledge on how classification-based machine learning works and have explained why semi-supervised learners are suitable for application in our problem domain due to their ability to be trained exclusively on data about normal system operation. Next, we have given an overview of how one-class classification can achieve anomaly detection and have explained why the particular algorithm we have chosen is a promising instance of this approach. We have briefly touched on issues that arise from the application of our approach in practice. In particular, we have explained how the highly dimensional space represented by multi-variate monitoring data makes it difficult for machine learning algorithms to learn an accurate model and stated that we aim to address this issue by finding suitable strategies for choosing subsets of metrics. Finally, we have discussed existing approaches that apply machine learning to achieve similar goals to ours and how our work differs from those.

What remains is to find out whether one-class classification can in fact achieve sufficiently accurate detection of failures based on instrumented system state and how we can improve its accuracy by effectively fragmenting the large feature space.

Chapter 7

Data-Driven Detection

In this chapter we present our evaluation of applying one-class classification to automate the detection of failures based on multi-variate monitoring data about system operation. We present three promising strategies to fragment the otherwise large feature space and select subsets of metrics. We reveal which of these strategies are suitable to achieve high levels of failure detection accuracy. We begin with an overview of the experiments and a description of their design and execution. We use monitoring data about the operation of our case study to train the one-class classification algorithm and describe the training and testing procedure we follow to estimate its accuracy. After explaining the set of accuracy measures, we evaluate the suitability of our evaluation method before presenting our results. We conclude this chapter by discussing the performance and limitations of data-driven detection and by presenting the insights we have gained about the accurate detection of system-level failures.

The key results we are able to report on in this chapter are as follows.

- We have found an automated fragmentation strategy that ranks attributes on their ability to form accurate decision trees and whose resulting attribute subsets support highly accurate failure detection in an automated manner.

- We present five highly accurate classifiers based on this fragmentation strategy. These classifiers detect more than 90% of failure instances and have false alarm rates that range from just one per seven hours to 11 per hour.

- We find that small subsets of attributes (two, three or four attributes) are sufficient to train highly accurate classifiers.

- Several fragmentation strategies that select attributes based on expert knowledge (i.e. per-host, metric category and per-component) do not result in highly accurate failure detection.

- However, the automtically selected attributes, which do lead to highly accurate failure detection, do no necessarily conform to what an expert might select based on experience with the task of failure detection.

7.1 Overview of Detection Approach

We want to automate the detection of failures by applying one-class classification to multi-variate data about the operation of a middleware-based service composition. This is in contrast to approaches that predict the occurrence of failures [Salfner et al., 2010]. We aim to establish whether a given set of measurements indicates that a failure is occurring at the time that these measurements were taken. We give a high-level overview of how this approach works and then discuss these aspects in more detail in Section 7.2.1.

We use an implementation of the one-class classification algorithm that we have described in the previous chapter as it is available in the WEKA [Machine Learning Group at University of Waikato, 2012] data mining workbench. This workbench allows us to set up experiments to assess the accuracy with which the algorithm can classify monitoring data as either describing normal or faulty operation of our case study. We describe the workbench and the parameters we have used in Section 7.2.5.

The input to the classification algorithm consists of about 100 minutes of monitoring data collected with our monitoring framework during operation of the Polymorph Search composition. This data consists of both data about normal operation and about faulty operation of the Polymorph Search composition, which serves as our case study (see Section 5.2). We have converted the raw monitoring data into a set of instances suitable for the classification algorithm as explained in Section 6.4. It is known which instances contain failures. We describe the training and test datasets in more detail in Section 7.2.2.

The algorithm is trained on instance data about normal operation and then tested on instance data about both normal operation and failures. During the training phase the algorithm is presented with a series of instances about normal operation, which provides an opportunity to learn a classifier of normal operation. During the testing phase the learned classifier model is then tested by presenting it with instances of both normal and faulty operation. For each instance the classifier outputs a classification, which indicates whether the instance describes normal operation or whether it is indicative of the occurrence of a failure. The WEKA workbench records the number of correct and incorrect classifications to estimate the accuracy of the classifier. We describe the measurement procedure and the resulting accuracy measures in more detail in Sections 7.2.3 and 7.2.4 respectively.

7.2 Experiment Setup

7.2.1 Overview

We begin with a high-level overview of the objectives of our evaluation and the corresponding experiments. We want to investigate whether one-class classification applied to multi-variate monitoring data about the operation of a service composition is a suitable mechanism to automate the detection of system-level failures. Our expectation is that a one-class classification algorithm can learn an accurate model of normal system operation based on such data. This model can then be used to identify abnormal monitoring data that is indicative of a failure. Closely related to this is the question of how we should fragment the feature space in order to support the one-class classification algorithm to learn an accurate classifier. As we have explained in the previous chapter in Section 6.4, the numerous and often continuous met-

rics that comprise our monitoring data form a large and multi-dimensional feature space. This makes learning an accurate classifier with a finite amount of training difficult and poses an obstacle to accurate failure detection based on one-class classification.

We address these two related questions with a controlled in-vitro experiment. We apply three fragmentation approaches to records of monitoring data in order to obtain different subsets of metrics. We use these to train an implementation of the one-class classification algorithm that we have presented in the previous chapter (Section 6.3). This results in three classifiers, each of which are able to differentiate between normal and abnormal system operation to varying extents. We measure the accuracy of these classifiers by testing them on monitoring data of normal operation and of failures that we have provoked using failure injection. This allows us to quantify the accuracy with which a one-class classifier can detect system-level failures, to compare the performance of the classifiers that result from the three approaches to fragment the feature space and thereby we learn how to approach the selection of subsets of metrics for use by the classifier so as to improve its ability to learn an accurate model.

The monitoring data for these experiments has been obtained by monitoring the operation of the Polymorph Search service composition with our prototype monitoring framework. We have used our failure injection tool to record monitoring data of system operation under failure. The case study has been described in Section 5.2 and the failure injection tool in Section 5.5.1.

The independent variable is the *fragmentation strategy* that is used to select subsets of metrics. It has three levels as defined by the three approaches to fragmentation that we use. We have chosen these approaches as each one appears to be promising while still adhering to our requirement to minimise the degree of manual intervention that is necessary to make the failure detection mechanism operational. The three fragmentation strategies do neither require a substantial degree of expertise with machine learning nor do they necessitate much preparatory data analysis or manipulation. The following list provides a basic overview of the strategies.

- **Metric Category.** In this approach, we select metrics based on their metric type. This fragmentation creates two subsets of metrics. One consists of resource usage metrics and the other one is based solely on performance indicators.

- **Per-Component.** With this approach, distinct subsets are formed from the metrics for individual components. Then, classifiers are trained to learn a model of normal behaviour on a per-component basis. We create subsets for four components.

- **Top-X.** In this strategy we let another algorithm automatically select subsets of metrics in order to optimise the accuracy of the internal classifier of the one-class classification algorithm. We use this approach to create differently-sized attribute subsets. Apart from determining the accuracy of these subsets, this allows us to examine the effect of the number of attributes on accuracy.

Our hypothesis is that there exist subsets of metrics that enable a one-class classifier to detect the occurrence of failures with high accuracy. And furthermore, that there exists a relatively simple strategy to select subsets that enable accurate classifiers. The outcome variable based on which we evaluate

our hypothesis is the *detection accuracy* that the classifiers achieve based on the subsets resulting from one of these fragmentations. The detection accuracy is determined by the rate of true positives and the rate of false positives the classifier achieves. The true positive rate (TPR) is the proportion of outliers (failures) that have been correctly classified as such out of all outliers (failures). More specifically, $TPR = \frac{TP}{(TP+FN)}$, where *TP* is the number of true positives or outliers (failures) correctly classified as such and *FN* is the number of false negatives or outliers (failures) incorrectly classified as normal (i.e. failures that the classifier missed to identify as such). TPR describes the ability of a classifier to detect failures. The false positive rate (FPR) is the proportion of normal instances that have been incorrectly classified as outliers (failures) out of all normal instances. More specifically, $FPR = \frac{FP}{(FP+TN)}$, where *FP* is the number of false positives or normal instances that have been falsely classified as outliers (failures) and *TN* is the number of true negatives or of normal instances that have been correctly classified as normal. FPR describes the rate of false alarms that are generated by a classifier. We explain how we assess and compare the detection accuracy of different classifiers based on their TPR and FPR in Section 7.2.4.

The null hypothesis states that the learned classifier in each case is too inaccurate to constitute a useful failure detection mechanism in practice. The alternative hypothesis states that the resulting accuracy is sufficiently high, so that the classifiers neither miss a high proportion of failures nor needlessly raise a large number of false alarms. More formally, we have:

$$H_0 : Accuracy_{DataDrivenDetection} < Practical \tag{7.1}$$

$$H_a : Accuracy_{DataDrivenDetection} > Practical \tag{7.2}$$

This set of hypotheses is not as clear-cut and the criteria for their evaluation—whether or not the detection accuracy could be sufficiently high for practical application—are rather subjective when compared to our investigation of the effects of data-driven diagnosis. Therefore, the following subsections explain all relevant aspects of our experiments in detail. We begin with an overview of the characteristics of the training and test set data that we use in the experiments. We then present the procedure we use to calculate the accuracy of classifiers followed by a discussion of the measures of accuracy and how they relate to failure detection. It is here that we clarify what we mean by "practical" or "highly accurate" failure detection. Finally, we explain how the experiments were implemented. The accuracy measures, their measurement procedure and the training and test data allow us to evaluate the hypothesis as objectively as is possible.

7.2.2 Training and Test Data

We have collected close to 100 minutes of monitoring data by using the Monere monitoring framework to observe the Polymorph Search service composition that we have described along with its deployment on our distributed testbed in Section 5.2. We have recorded three types of operation: normal operation, operation under the occurrence of failures and quiescent operation, during which all components are operational, but not actively involved in processing any requests. We use measurements of the components of only a single host from our case study. We have selected host *huey* in the UCL domain for this purpose

as all injected failures either directly impact or propagate their effects to at least one component on this host. We use measurements of only one host in order to limit the number of attributes that need to be considered by the classifier and to improve the tractability of the feature space. Even on this individual host there are hundreds of metrics that lead to instances with numerous metrics. The monitoring data we have collected is summarised in Table 7.1.

Table 7.1: Overview of the monitoring data we have recorded to form the overall training and test sets for the classifiers.

Type	**Time**
Quiescent	1,290s
Workflow runs	2,820s
Failure 1	150s
Failure 2	320s
Failure 3	370s
Failure 4	350s
Failure 5	260s
Failure 6	350s

Data about normal operation was collected based on four executions of the Polymorph Search composition. All four executions were performed on the same input data set and configured to run for one molecule packing type (i.e. one MOLPAK execution) that then triggered 100 parallel invocations of the dmarel BPEL sub-process. During the time of these executions, there were about 30 Grid compute nodes available and the average duration of compute jobs was approximately 170 seconds. We have recorded 47 minutes of normal operation in this way. Similarly, we have obtained 21 minutes and 30 seconds of quiescent operation (i.e. all components are operational, but the application is not being executed on any requests). This gives a total of 68 minutes and 30 seconds of normal operation.

We used our failure injection tool (cf. Section 5.5.1) in order to be able to monitor system operation under certain failures. For each failure, we collected data for the period starting at about 10 seconds before the injection and up until a few minutes after the injection had activated the failure. This increases the number of test instances we have available and allows us to present the classifier with a non-trivial test set[1]. This means that we test our failure detection mechanism on failures of long temporal extent and our results may not apply to transient and rather short-lived ones. However, as we are interested in the detection of failures that need to be diagnosed and repaired, before their effects can be remedied, this is not a serious limitation in our case.

The six failures are the same ones we have used for the diagnosis experiment. We list them here as a reminder.

1. unavailability of a remote service without known cause

[1] A test set with a very small number of non-target class instances is trivial as a classifier that guesses only the target class could result in an optimistically positive accuracy estimate.

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

These failures exhibit symptoms, such as abnormal application termination, unexplained application slowdown at various stages of the application and missing results. Previously, we have established that human system operators are able to diagnose the causes for these failures given the monitoring data collected by Monere. Hence, we know that the collected data exhibits the symptoms that allow the recognition of these failures. Failure 1, for which the remote service is unavailable is visible through the availability metrics of the Web service. Failure 2, during which the remote Web service experiences a drastic slowdown, is visible through a substantially higher execution time metric recorded by its clients on host *huey*. In Failure 3, the Tomcat JVM on host *huey* exhausts the number of threads assigned to it by the operating system. This is visible through a high thread count metric and through the availability metrics of the JVM. Similarly, in Failure 4, the same Tomcat JVM exhausts its heap space. The values for the heap usage metric are somewhat higher than usual. In Failure 5, Condor is unable to schedule jobs as it is not able to store the temporary files it needs for this purpose on the local file system. This failure is visible through the used percentage metric of the local file system. In Failure 6, the Condor scheduler has received requests for a large number of compute jobs by the time the Polymorph composition submits its first compute jobs. This significantly increases the time jobs spent waiting in the queue. One set of metrics through which this failure becomes visible is the number of idle jobs in the queue and the number of all available compute nodes. These are just some of the more obvious metrics as they have also often been used by the participants of our previous experiment to diagnose these failures. It is possible and quite likely that several other metrics are well-correlated with these failures.

7.2.3 Measurement Procedure

Each classifier is trained and tested according to a particular procedure in order to calculate its accuracy. In general, a classifier is trained on a certain number of instances, which form the so-called *training set*, and is subsequently tested on a separate set of instances that have not been encountered by the classifier during training and that is known as the *test set*. Given that we only have access to a limited amount of data, there is a tension between the size of these two sets. In order to have an opportunity to learn an accurate classifier, we want to maximise the number of instances in the training set. However, we also need a large number of instances to be able to arrive at realistic estimates of the accuracy of the trained classifier.

We rely on an approach that is commonly used for the performance evaluation of classifiers in machine learning. *stratified 10-fold cross-validation* has been found to produce reliable performance estimates of classifiers in practice in cases where the size of the available data set for training and testing

is limited [Witten and Frank, 2005, page 150]. It makes good use of the available data by allowing us to train a classifier on as much data as possible and then train it on different test sets. It reshuffles training and test data several times, which ensures variability of the data. These characteristics increase our confidence that the accuracy measure we obtain in this manner are close to what we would observe from the application of the failure detectors in practice.

10-fold cross-validation works as follows for standard two-class and multi-class classifiers. First, the entire data set is split into 10 stratified folds. The individual instances are sampled randomly in such a manner that each resulting fold contains approximately the same proportion of each class as in the overall data set. This is what makes it stratified. Next, the classifier is trained on nine folds containing this mix of classes and tested on the remaining one. Once the first round of accuracy measures has been calculated in this manner, another fold is selected for testing and the previous test fold is added to the other eight training folds. This process is repeated until every fold has been used for testing once and the individual accuracy measures are averaged together. In order to increase the reliability of the resulting measures, this process can itself be repeated 10 times, which is then referred to as 10-times 10-fold cross-validation.

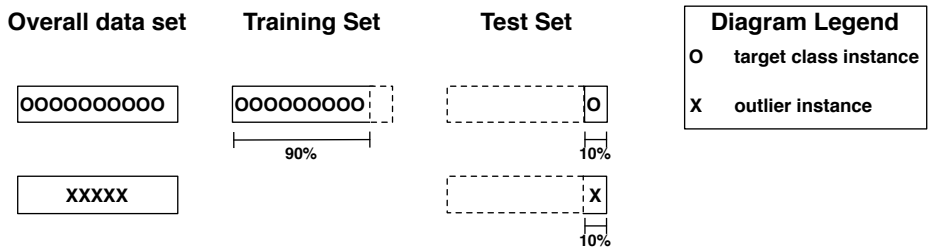


Figure 7.1: During cross-validation of a one-class classifier it is trained on 90% of target class data and tested on the remaining 10% and on 10% of non-target class data.

When cross-validation is applied to train and test a one-class classifier, folds are created in a slightly different manner as is illustrated in Figure 7.1. Training is performed on 90% of the target class (normal instances). Testing is performed on the remaining 10% of the target class instances and on 10% of randomly selected outlier instances. Each test set contains only 10% of non-target class instances to ensure that the proportion of both classes is approximately the same. As is the case for multi-class classifiers, the folds are then swapped ten times, the accuracy measures are calculated each time and eventually averaged. Variability is achieved by reshuffling the folds between repetitions of the cross-validations.

We add a step to the standard approach in order to obtain learning curves. A learning curve graphs the performance of a classifier observed under exposure to differently sized training sets. Figure 7.2 shows an example of the kind of learning curve we use. We plot the Kappa statistic (y-axis) achieved at a given training set size against the number of instances used for training (x-axis). The first data point shows the results of using 10% of the training and test set instances. The y-axis usually represents the

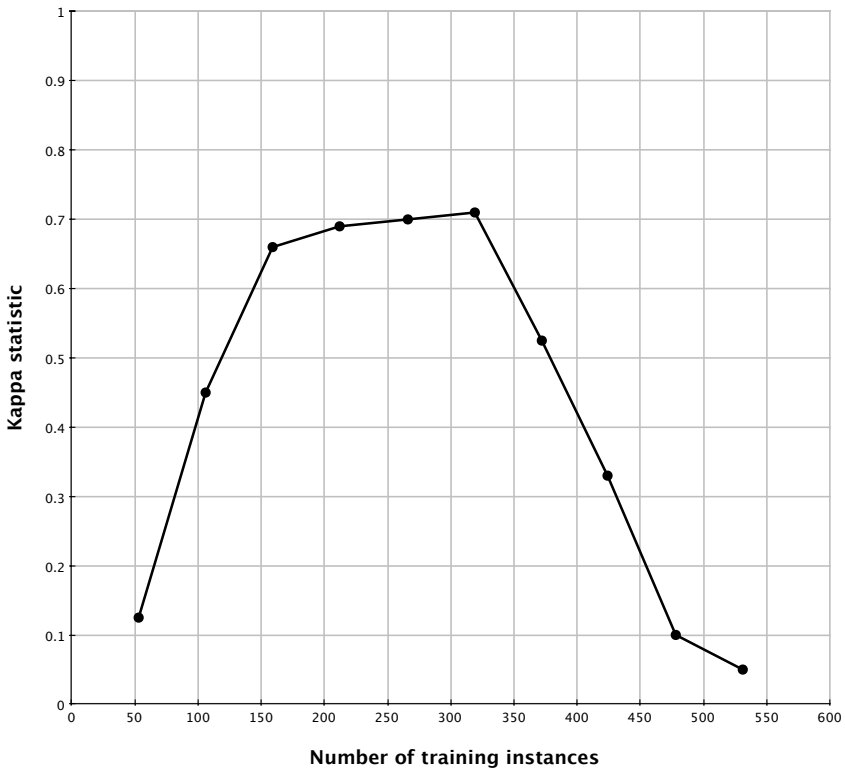


Figure 7.2: Example of a learning curve that plots the Kappa statistic of a classifier against the training set size at which it has been observed.

proportion of correctly classified instances, but we prefer the Kappa statistic as it describes how much better a classifier performs compared to a random one.

Being able to inspect the effect of the training set size on the resulting accuracy of the classifier is useful. As we have explained in the previous chapter, in a highly dimensional feature space a great amount of training data may be needed so that the classification algorithm can learn a good classifier. However, the performance of a classifier may actually begin to decrease again once a certain amount of training data has been reached. With the help of a learning curve, we can determine whether any such saturation is taking place and select the training set size at which a given classifier reaches its peak performance. In some cases, this may happen with 100% of the available training instances being used and at other times a classifier may reach its peak much sooner and has its performance degrade as the size of the training set increases further. While we cannot determine exactly how the performance would change with significantly more training data than what we have available, plotting learning curves in this manner enables us to detect when saturation has taken place or when further improvement seems possible.

The procedure we use in order to obtain learning curves is as follows. We split the overall data set into 10% subsets. We then perform stratified 10-fold cross-validation on this subset, before generating another 10% subset from the overall dataset and repeat this process 10 times per subset size. We then

proceed in the same manner with 20%, 30% subsets and so on until we train and evaluate the mechanism on 100% of the available instances. We perform ten 10-fold cross-validations per subset size and each of the ten repetitions makes use of a reshuffled subset. Testing is also performed on a corresponding 10% subset. The accuracy metrics are averaged from the individual runs per subset size.

7.2.4 Measures of Accuracy

We quantify the failure detection accuracy of our classifiers through a set of measures that are commonly used in the evaluation of machine learning classifiers. These measures are as follows.

- The **Kappa statistic** compares the number of correct classifications achieved by the classifier that is under examination to the number of correct classifications that would be achieved by chance and counts only any extra successes [Cohen, 1960].

- The **true positive rate (TPR)** is the proportion of correctly classified outliers out of all outliers. In our context, outliers are synonymous with failures. More specifically, $TPR = \frac{TP}{(TP+FN)}$, where *TP* is the number of true positives or outliers (failures) correctly classified as such and *FN* is the number of false negatives or outliers (failures) incorrectly classified as normal (i.e. failures that the classifier missed to identify as such). TPR describes the ability of a classifier to detect failures.

- The **false positive rate (FPR)** is the proportion of normal instances incorrectly classified as outliers or as failures out of all normal instances. More specifically, $FPR = \frac{FP}{(FP+TN)}$, where *FP* is the number of false positives or normal instances that have been falsely classified as outliers (failures) and *TN* is the number of true negatives or of normal instances that have been correctly classified as normal. FPR describes the rate of false alarms that are generated by a classifier.

Two concepts that are useful for the evaluation of a failure detection mechanism are its *sensitivity* and its *specificity*. In medical applications, the sensitivity of a diagnostic process describes its ability to correctly identify diseased patients. It is equivalent to the TPR and at 100% of sensitivity, our mechanism would not miss to detect any failures. Specificity refers to the ability of a diagnostic process to avoid classifying healthy patients as diseased. It is equal to $1 - FPR$ and at 100% of specificity, which corresponds to an FPR of 0%, our mechanism would not raise any false alarms. The higher the sensitivity and specificity of a classifier, the more accurately it can detect failures.

The Receiver Operating Characteristic (ROC), which was originally applied to represent the trade-off between hit rate and false alarm rate in signal detection [Green and Swets, 1966], can be used to illustrate the performance of a classifier in terms of its sensitivity and specificity. Figure 7.3 shows an example of a ROC coordinate space. The x-axis measures the FPR or $1 - specificity$. We prefer smaller values along this axis as they represent smaller rates of false positives (i.e. fewer needless failure alerts). The y-axis represents the TPR or sensitivity. We prefer higher values along the y-axis as this denotes a mechanism that correctly detects a higher rate of failures or outliers.

In common usage, a so-called ROC curve is obtained by varying a threshold parameter for the classifier, such as its sensitivity (x-axis). The performance of different classifiers can be compared by

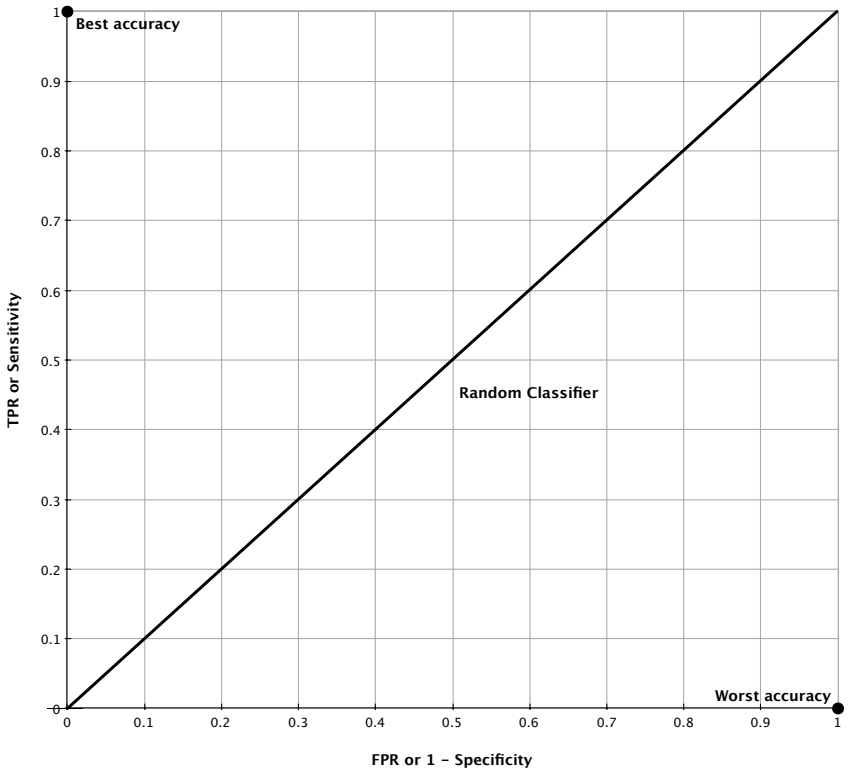


Figure 7.3: The coordinate space of the Receiver Operating Characteristic (ROC) used to measure and compare the detection accuracy of our classifiers.

calculating the Area Under the Curve (AUC) for each classifier. The "curve" for a random classifier that guesses one or the other classification with a probability of 50% each is shown as a diagnoal line in Figure 7.3. It starts at point (0, 0) in the lower left corner of the space and extends through to point (1, 1) in the upper right-hand corner. The AUC of the random classifier is always 0.5 and any classifier that performs better than the random one has a greater AUC.

We do not manipulate the sensitivity to obtain a ROC curve for a number of reasons. First and foremost, our definition of what constitutes a highly accurate failure detection mechanism (explained soon), removes most regions of the ROC space from consideration in any case, as the last paragraph in this section explains. Second, the use of the AUC to compare the performance of classifiers in a fair manner has recently been called into question [Hand, 2009] as the resulting AUC depends on the classifier that is evaluated and may vary from classifier to classifier hence making an objective comparison more difficult. Third, we do not have detailed knowledge of the cost of either type of mis-classification, which would allow us to select an optimal trade-off between TPR and FPR.

Instead, we compare the performance of classifiers with a slightly different approach; by comparing points in the ROC coordinate space. As explained in the previous section, we test each classifier at varying training set sizes, which results in a set of accuracy measures. Among those measures are the values for TPR (sensitivity) and FPR ($1 - specificity$) that a classifier was able to achieve. We

select the TPR-FPR pair that represents the highest accuracy for an individual classifier and compare the resulting points in ROC space occupied by different classifiers. In general, the "best" TPR-FPR pair for a classifier corresponds with the training set size for which the classifier has achieved its peak Kappa statistic. Points above and to the left of the line formed by the random classifier are preferable as they represent higher levels of overall accuracy than do points that are below and to the right of this line. The highest accuracy is represented by point (0, 1), which is in the top left-most quadrant in the ROC space. This point signifies a classifier that correctly detects all failures and does not raise any false alarms. The worst possible accuracy is represented by point (1, 0), which denotes a failure detector that misses all failures and mis-classifies all instances of normal operation as failures.

Fault detection is a cost-sensitive problem. The points in ROC space represent trade-offs between the TPR and FPR of a classifier. The optimal point of this trade-off depends on the cost of false positives and of false negatives. While it may be possible to estimate the approximate ratio between the two, the exact cost needs to be determined on a case-by-case basis. In general, we consider missed failure occurrences to carry a higher cost than false alarms. Missing to detect a failure as quickly as possible creates an opportunity for it to propagate unnoticed for a longer period of time, during which it can continue to cause an outage. Needlessly alerting operators to a non-existent failure, wastes their effort, may reduce their trust in the mechanism and may cause them to become reluctant to act on its alerts. While it is important to minimise both of these mistakes, the former leads to a cost that is directly measurable in terms of increased downtime and has effects that are visible by end users. Therefore, we prefer classifiers that achieve a higher rate of sensitivity or TPR even, to some extent, at the expense of FPR.

We consider a classifier to be *highly accurate* based on its TPR and FPR. We regard it as being promising to support accurate failure detection in practice, if its TPR is at least 90% and its FPR is at most 3%. This translates to a failure detection mechanism that correctly alerts operators to at least nine out of ten failures. Furthermore, consider that each instance a classifier is presented with for analysis represents a ten-second window of monitoring data. At an FPR of 3%, it would incorrectly classify about 3 out of 100 normal instances as failures. That is, it would needlessly alert a system operator to a non-existent failure approximately eleven times per hour. The time wasted on false alarms will vary on a case-by-case basis, but we assume this to be a worst-case false positive rate that could still be sustainable. We cannot impose a complete ordering on the TPR-FPR pairs, as this would require us to know how much more important one is compared to the other. However, we can compare pairs by specifying that, within the set of classifiers that meet the two criteria for high accuracy, we prefer those that maximise their TPR at the expense of FPR within the limits prescribed here (i.e. $FPR \leq 3\%$). Consequently, we compare points that lie within a small region of the ROC space (close to point (0, 1)) and disregard all other classifiers.

7.2.5 Learning Workbench

We use the WEKA [Machine Learning Group at University of Waikato, 2012] data mining workbench in order to run the experiments with the one-class classification algorithm. The WEKA workbench provides

implementations of numerous machine learning algorithms and facilitates experimentation with them. It is possible to set up experiments in terms of the instance data that is to be used and to configure evaluation procedures that are to be executed, such as 10-fold cross-validation on randomly composed data subsets of different sizes.

We briefly describe the configuration of the algorithms we use in order to support independent verification of our results through repetitions of the experiment. The one-class classification algorithm relies on an internal classifier in order to learn a discriminator between the target class and the outlier class represented by the artificial data (cf. 6.3). We have selected a decision tree that is based on the C4.5 [Quinlan, 1993] algorithm as the internal classifier. We have compared the performance of Bayesian Networks and of decision trees as internal classifiers and found the latter to achieve considerably better accuracy at a lower computational overhead. The implementation of the C4.5 algorithm in the WEKA workbench supports continuous and discrete attributes and can also handle missing attribute values.

We have used the default parameters as they are suggested by the implementations in WEKA of the C4.5 and of the one-class classification algorithm. After testing several configurations of these parameters, we have not found any substantial improvements to the accuracy of the classifier. The one-class classifier has been configured to use a Gaussian generator with a mean of 0 and a standard deviation of 1 for use in density estimation on the target class data. The decision tree algorithm has been configured to apply post-pruning in order to improve its accuracy.

7.3 Evaluation Method

The intrinsic (does the object of our investigation work?) and its extrinsic evaluation (how does it behave?) of data-driven failure detection are performed through the same set of experiments. The goal of these experiments is to determine, first, whether one-class classification can learn to discriminate between normal and failure operation based on multi-variate monitoring data with high accuracy and second, what levels of accuracy are attainable given different strategies to fragment an otherwise large feature space. There are various options with which to investigate these questions.

A set of field studies would provide us with the greatest confidence in the representativeness of our results. As part of a field study we would have had to integrate the failure detection mechanism into a suitable monitoring framework that observes a service composition in production use. This would have enabled us to quantify the failure detection accuracy under fully realistic circumstances, which may reveal unexpected challenges. Furthermore, this would have created an opportunity to learn how system operators interact with such a mechanism. However, a field study remains an expensive endeavour that necessitates access to service compositions in production use and their personnel for potentially several months in order to be able to observe the performance of the mechanism under a wide variety of conditions and failures. This is not feasible as part of this thesis.

Simulation occupies the low end of the spectrum of possible evaluation approaches that we could have used in this case. With simulation we would have generated synthetic data to simulate various aspects of the operation of a service composition under normal and failure conditions according to some

analytic model that we would have prepared for this purpose. The benefits of such an approach are that it would allow us to test the mechanism on data that, purportedly, describe a wide variety of conditions. This in turn could help us to more thoroughly explore the limits of our failure detection mechanism. Apart from not necessarily being easy to implement, the main disadvantage of such an approach is that it is rather difficult to ensure that our synthetic data is really representative of system operation and of failures as they occur in practice.

As before, our approach is a compromise between these two extremes. Our controlled experiment is based on monitoring data that we have obtained from a replica of a real-world service composition and not on a synthetic profile of the operation of service compositions. We provoke a set of failures that have been observed to occur in production use by injecting the conditions that lead to these failures. However, this approach cannot guarantee that it includes all issues that may occur under real operation in the field over a long period of time. There may be aspects that we are unaware of and that are not reproduced by our approach. Nevertheless, this experimental setup allows us to perform repeated controlled experiments on largely realistic data. The outcome measures we obtain in this manner afford a comparison of the different classifiers and their evaluation against the criteria we have imposed for highly accurate failure detection. Overall, our evaluation approach represents a reasonable trade-off between the cost of the experimental setup and the representativeness of its results. It is more affordable, especially when considering our limited resources, than a set of field studies, yet provides us with more confidence that the measured accuracy levels come close to what the mechanism would achieve in the field than a simulation-based approach could.

7.4 Accuracy of Data-Driven Detection

It is necessary to fragment the large feature space that is represented by our monitoring data in order to enable the classification algorithm to learn an accurate classifier. We have explained the reasons for this in Section 6.4. We examine the effect three fragmentation strategies have on the detection accuracy of the resulting classifiers. First, we select attributes according to the type of metric they represent. We compare an attribute set that consists of only resource usage metrics to one based exclusively on performance indicator metrics. Next, we measure the detection accuracy for classifiers based on attributes that represent the metrics of individual components. Finally, we quantify the detection accuracy achieved based on attribute sets that are selected automatically in such a manner as to optimise the accuracy of the internal classifier (i.e. decision tree). Within this experiment, we examine the effect of the number attributes in the resulting subset on detection accuracy. In each of the following subsections, we describe and motivate one of the fragmentation strategies and present the accuracy measures of the resulting classifiers.

We also measure the execution times of most classifiers. For this purpose, we use the wall clock time of training and testing a classifier at 100% of instances. This allows us to compare the times required by the different classifiers for processing the same amount of data and represents a kind of worst-case result within the available data. The measurements were performed by the WEKA workbench through time stamps taken at the beginning and ends of each training and testing phase in its Java implementation. The

times measured for testing represent the time a classifier takes to process and classify about 60 instances, which together represent 10 minutes of system operation.

Before we begin with presenting these results, we examine the attribute set that contains all metrics for host *huey*. We do this in order to find out whether the Hughes Effect is in fact in effect in our case and to demonstrate the need for fragmentations to obtain smaller subsets of attributes.

7.4.1 Per-Host Classifier

We begin by examining the attribute set that consists of all metrics for the components of a single host. The resulting classifier can detect failures on a per-host basis. We do not expect this fragmentation to lead to highly accurate failure detection, but instead examine it in order to demonstrate the need for an approach that leads to more useful attribute subsets.

We refer to the resulting subset, which is based on the metrics of all components on host *huey*, as *attribute set A*. In order to form this attribute set, we have removed common distracting attributes. These include the time stamp of an instance, the number of times a component has been found to be unavailable and attributes representing metrics of the measurement subsystem of a monitoring agent. Part of the latter are metrics such as the number of failed measurement collections an agent has recently experienced. This could make detection trivial by revealing failures for which some of the monitored components have become unavailable.

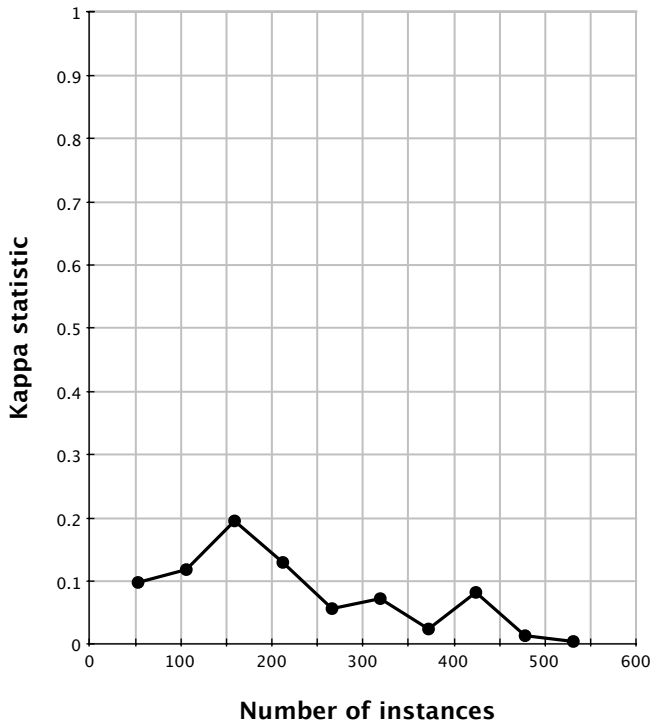


Figure 7.4: Learning curve for the per-host classifier using its achieved Kappa statistic as the performance measure.

In Figure 7.4 we show the learning curve for attribute set A. It graphs the Kappa statistic that the classifier achieves against the number of training instances used. Table 7.2 shows a summary of its performance in terms of the peak Kappa statistic, at which training set size it was achieved and the corresponding FPR and TPR of the classifier. The classifier based on attribute set A achieves only marginally better accuracy than a random predictor when using the full training set (i.e. 100% of training instances). It reaches its peak accuracy at 30% of training instances. This may merely represent a local maxima that can be improved with considerably more training data, but the learning curve does not reveal any such trend. Even at its peak, classifier A will classify slightly less than $1/5$ of failures correctly, while still mis-classifying 1 in 100 normal instances as a failure. On average, a single cross-validation run on the entire attribute set A takes 0.0284 seconds for training the classifier and 0.0002 seconds for testing it. The latter figure is the time classifier A takes to process and classify about 60 instances of monitoring data.

Table 7.2: Summary of classifier performance for attribute set A (all metrics).

Set	Instances	Kappa	FPR	TPR
A	30%	0.196	0.010	0.177

These results confirm that the feature space formed by the monitoring data of even a single host is indeed too large for the one-class classification algorithm to learn an accurate classifier based on 60 to 70 minutes of training data[2].

7.4.2 Metric Category Classifiers

This strategy creates attribute subsets that are based on metrics of a certain type. The idea behind this approach is to mimic the way in which a system operator could approach the task of detection by examining certain types of metrics in isolation as they can reveal particular kinds of issues. We have created two attribute sets according to this approach. One contains only attributes for resource usage metrics and the other consists of performance metrics. While a classifier based on only one type of metric may not be able to detect issues that are more visible in other types, a series of such classifiers for different types could achieve good failure detection coverage. Failures may not be exclusively observable through one set of metrics. In our set of failures, Failures 3, 4 and 5 are more readily detectable through resource usage metrics; Failures 2 and 6 are more prominent in the available performance metrics. And Failure 1 is an unavailability issue, which is not directly captured in any of the metrics represented in these two attribute sets. In this instance, we have not modified the test sets accordingly as all failures may have some potentially discernible effect on each type of metric.

Attribute set B contains 82 attributes that correspond solely to resource usage metrics. Examples include the number of threads, file descriptors and memory used by an operating system process, file system usage metrics, the amount of physical memory used and CPU load statistics. The other subset is *attribute set C*, which only includes performance indicator attributes, such as the number of disk reads,

[2]The classifier is trained on at most 90% of normal instances (i.e. nine training folds composed from the overall attribute set).

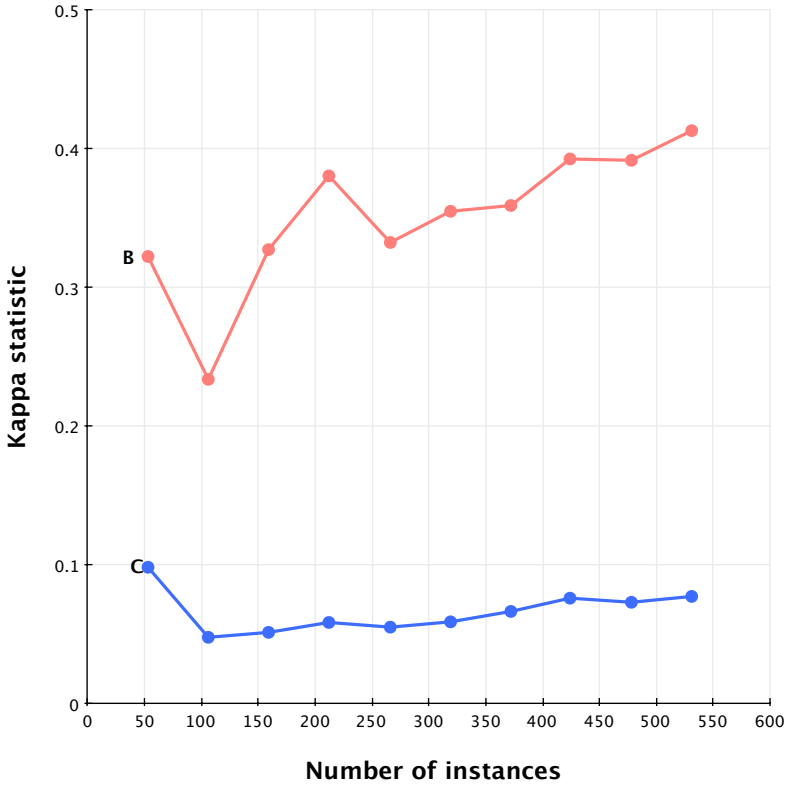


Figure 7.5: Learning curves for the classifier based on resource usage metrics (B) and on performance indicators (C).

the number of transmitted and received packets at a network interface or the number of Grid compute jobs that are running and their durations. This set consists of 24 attributes.

Table 7.3: Comparison of performance of metric category classifiers.

Set	Instances	Kappa	FPR	TPR
B	100%	0.413	0	0.340
C	10%	0.099	0.010	0.09

The classifier based on attribute set B leads to a higher detection accuracy than the one based on set C. The former reaches its peak accuracy at 100% of training instances and its learning curve, shown in Figure 7.5, suggests that a further improvement at higher numbers of training instances may be possible. At its peak performance, the classifier based on set B classifies $1/3$ of failure instances correctly and does not mis-classify any normal instances. These values suggest that its hypersphere is too wide. That is, even though it includes all normal instances, a great number of failure instances fall within the boundary that encompasses normal ones. The resulting hypersphere admits $2/3$ of failure instances. As such, the performance of this classifier does not meet our requirements for highly accurate failure detection.

The classifier based on attribute set C achieves even lower detection accuracy. The learning curve

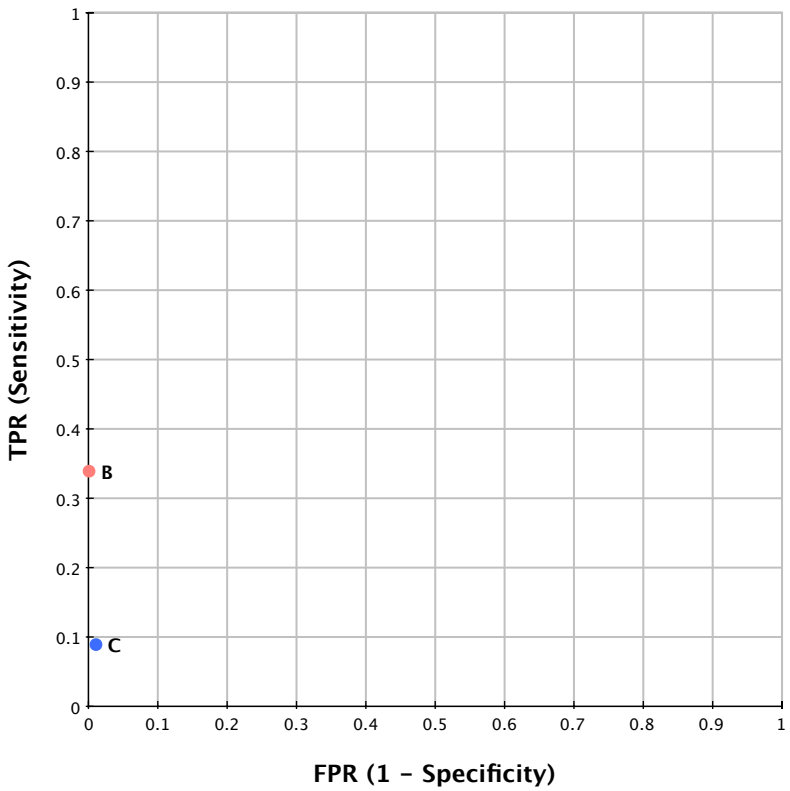


Figure 7.6: The TPR and FPR of the metric category classifiers within the ROC coordinate space.

for set C in Figure 7.5 reveals that it reaches its peak performance at only 10% of training instances. Even then, it only classifies 9 in 100 failure instances correctly. Its performance may increase with a considerably larger training set, but this classifier would need to significant improvement, before it could become useful for failure detection. Table 7.3 compares the accuracy of the two classifiers and Figure 7.6 depicts their peak accuracy in the ROC coordinate space.

On average, a single cross-validation run for the entire attribute set B takes 0.4798 seconds for training and 0.0009 seconds for testing. For the less accurate attribute set C, these times become 0.1144 seconds and 0.0003 seconds respectively. This represents a considerable difference between the two classifiers. However, in practice the times spent on processing instances are small in both cases[3].

These results lead to a few insights and raise some further questions. The metrics in attribute set B are affected by three of the six failures, instead of just two as is the case for set C. However, this may not be sufficient to explain the difference in the accuracy of the resulting classifiers. Furthermore, even though set C has a considerably fewer attributes than set B, the number of attributes by itself does not appear to be a good predictor of accuracy. This observation may indicate that there is a certain number of attributes that provides for the necessary variety of observations, while still avoiding to overwhelm the classification algorithm's ability to discern structure from the examples. We examine the effect of the number of attributes on accuracy in Section 7.4.4.

[3]Test testing figures represent the time to process about 60 instances that together equal 10 minutes of system operation.

The classifier that is based on resource usage metrics represents an improvement over the unfragmented set (attribute set A). However, it cannot even detect 50% of failures. The accuracy of the two classifiers is generally so low, that even their combined use is unlikely to lead to acceptable results. A naïve fragmentation of the attribute set based on metric type may not be able to provide for a level of accuracy that would make such a mechanism useful for failure detection in practice. This may not be surprising, as it is not entirely clear how a system operator selects worthwhile subspaces out of the overall problem space to focus on for failure detection. From our results, it does not seem worthwhile to continue to experiment with additional subsets chosen according to metric type. We can conclude that resource usage and performance metrics do not serve as the basis for a good classifier.

7.4.3 Per-Component Classifiers

As a second approach to reduce the dimensionality of the feature space, we group attributes by the components whose metrics they represent. The classification algorithm is only presented with the metrics of a particular component and learns to detect failures per component. Together, the components we have selected to test this strategy are affected by all six failures and represent a range of different component types, such as a file system, a dependency on a Web service in another administrative domain, aspects of the Tomcat JVM and an operating system process. All failures do not affect each of these components individually. Therefore, we have modified the test set for each classifier to contain only failures that its corresponding component is subject to. If it was successful, this approach would have the added benefit or providing some localisation of failures. Table 7.4 summarises the accuracy of the per-component classifiers. In some cases, especially when the overall accuracy is relatively low, the point at which a classifier achieves its peak Kappa statistic does not coincide with its best accuracy. The learning curves in Figure 7.8 do not seem to suggest that the availability of additional training instances would lead to considerably higher accuracy.

Table 7.4: The data points for per-component classifiers at their peak Kappa statistic and for the best accuracy in terms of TPR and FPR.

Set	Instances	Kappa	FPR	TPR
file system	100%	0.012	0.657	0.77
	10%	0.534	0.032	0
IDC	50%	0	1	1
	10%	0.62	0.005	0
condor_schedd	100%	0.392	0	0.283
	30%	0.390	0.1	0.438
Tomcat CL	100%	0.617	0.206	0.749
	20%	0.844	0.012	0.775

The first per-component attribute set represents the local *file system* on host *huey* and consists of 12 attributes, such as uptime, downtime, disk reads and writes per minute as well as the free and used disk space. Several other components rely on the file system for part of their functionality and as such this is

appears to be a promising attribute set for our purposes. Failure 5, the exhaustion of disk space on the local file system, directly affects this component and its instances accordingly constitute the test set for this classifier.

At 100% of training instances, the file system classifier achieves a fairly high TPR by correctly identifying more than $3/4$ of failure instances. However it also classifies two thirds of normal instances as failures. In this case, the hypersphere is simply too narrow. The file system classifier achieves its peak Kappa statistic at 10%[4]. The classifier has a TPR of 0 at this point while still mis-classifying one third of normal instances as failures. The elapsed wall clock time for training the file system classifier on 100% of the training set is 0.1053 seconds and the time for testing (i.e. process and classify about 60 instances) was measured to be 0.0003 seconds. File system metrics do not appear to afford a good classifier.

The next attribute consists of the metrics of an *inter-domain component (IDC)*, which represents a dependency from clients on host *huey* onto the Web service in the remote administrative domain. Its 18 attributes represent various metrics, such as availability, request rate, the latency between the client host and the Web service host and execution time metrics for the operations invoked at the remote Web service. The remote Web service that is represented by this inter-domain dependency is affected by Failures 1 and 2, which cause it to become unavailable (Failure 1) or overloaded (Failure 2).

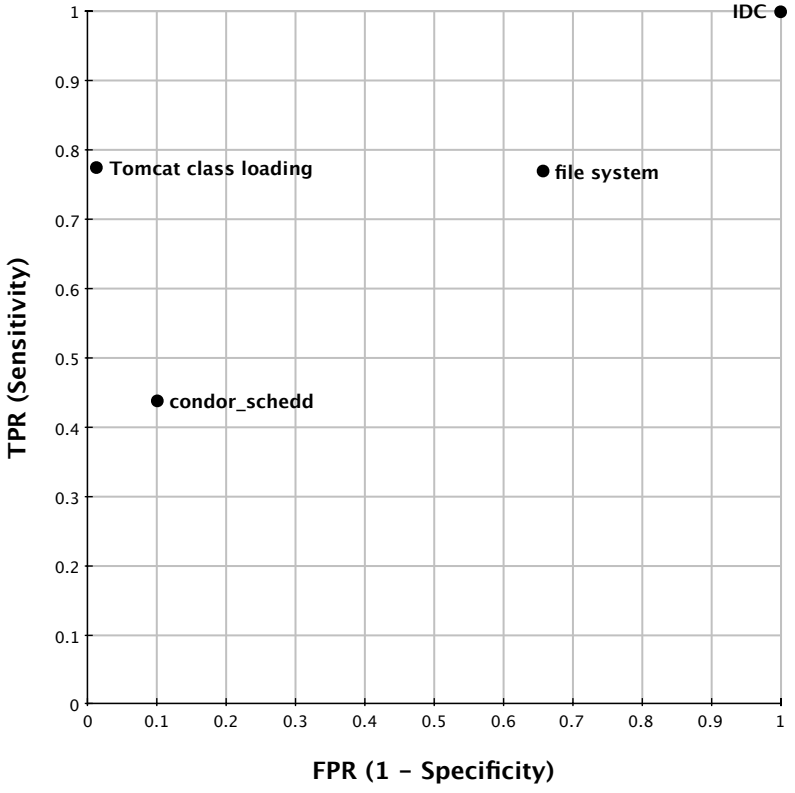


Figure 7.7: The TPR and FPR of the per-component classifiers within the ROC coordinate space.

The classifier resulting from the IDC set oscillates between classifying most instances as failures

[4]This is one of the cases in which peak Kappa statistic does not coincide with the most favourable TPR and FPR values.

or as normal. Everything is either a failure, or conversely no instances represent failures. The peak Kappa statistic at 10% of instances is a consequence of the classifier incorrectly classifying the majority of instances as normal. This represents a good guess as most instances in this problem domain tend to be normal. The average time for a single cross-validation run on 100% of instances takes 0.1875 seconds for training and 0.0003 seconds for testing it. Approaches based on simple heuristics are likely to outperform the IDC classifier for the two relevant failures. For example, a mechanism that triggers a failure alert in case the remote Web service has been recorded as being unavailable for two or three consecutive collection intervals or a mechanism that compares observed execution times with threshold values would probably perform significantly better at the detection of these failures.

The third attribute set contains the metrics of an *operating system process*. The condor_schedd daemon process is part of the Condor Grid job scheduler. The attributes represent availability metrics, the current number of threads and file descriptors held by the process, CPU utilisation and physical as well as virtual memory utilisation. The Condor process is affected by Failures 5 (disk space exhaustion) and 6 (overloaded Grid job queue).

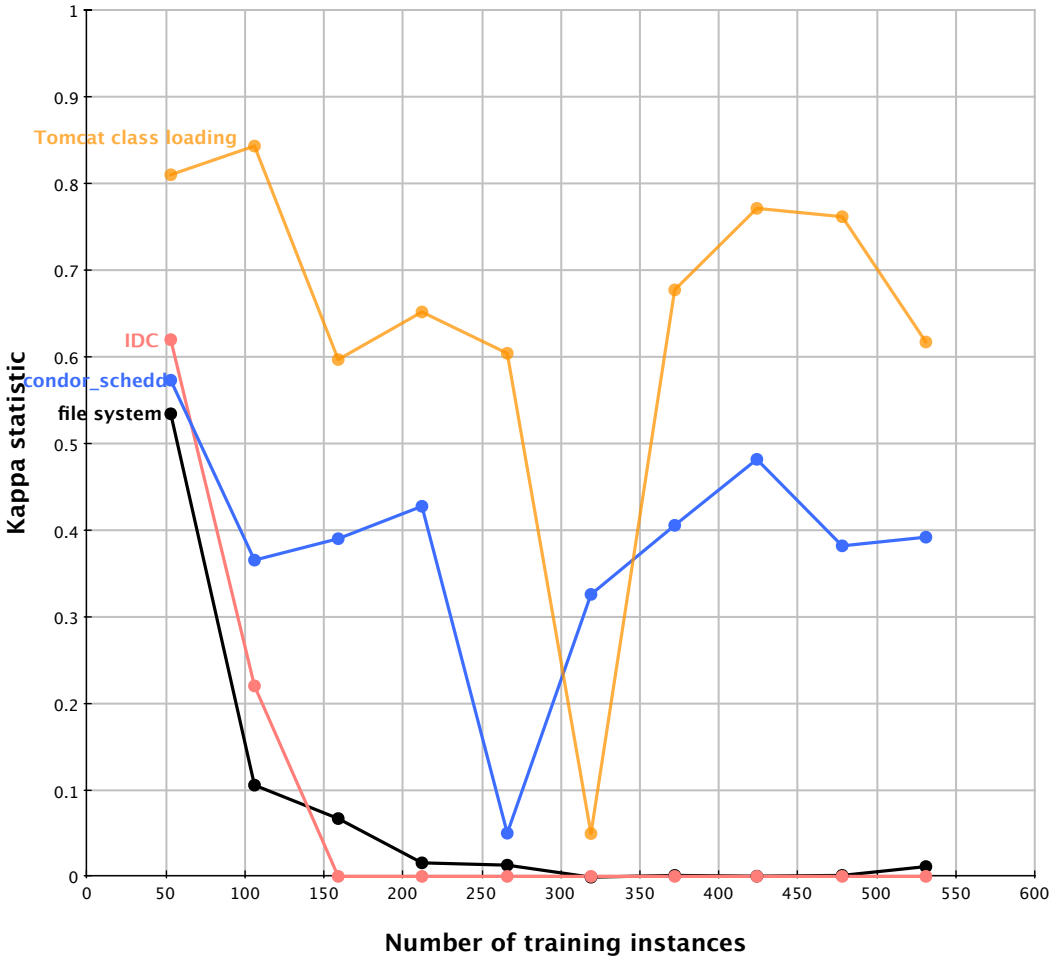


Figure 7.8: Learning curves for the per-component classifiers.

The condor_schedd classifier achieves its highest accuracy (but not peak Kappa statistic) at 30% of instances. The classifier detects slightly over 40% of failures correctly, while it mis-classifies 10% of normal instances as failures. On average, training of this classifier at 100% of instances takes 0.0764 seconds and the classifier needs 0.0002 seconds to process and classify about 60 instances. Its accuracy is still far from what we consider to be useful in practice. However, it represents an improvement over the previous two attribute sets. One difference to the previous two sets may be that the condor_schedd set contains a wider variety of metrics. It measures the usage of various resources, such as CPU, memory, threads and file descriptors.

The fourth and final per-component attribute set consists of 7 metrics of the *Tomcat Class Loading System (Tomcat CL)*. The attributes include the number of loaded and unloaded classes, the rate at which classes have been loaded and some availability metrics. As a subsystem of the Tomcat JVM, the Tomcat CL is primarily affected by Failures 3 and 4, which are the thread and the heap exhaustion by the Tomcat JVM process.

The classifier for the Tomcat CL attribute set achieves the highest accuracy of all per-component classifiers that we consider. Its best accuracy coincides with its peak Kappa statistic and is reached at 20% of training instances. It manages to correctly identify 77.5% of failure instances while only mis-classifying 12 out of 100 normal instances as failures. As such, it at least comes close to our criteria for highly accurate failure detection. The training curve for this classifier, shown in Figure 7.8, does not suggest that additional training instances may lead to better accuracy. Instead, its accuracy has degraded somewhat at 100%. Training and testing of this classifier takes 0.0532 and 0.0001 seconds on average

The first two per-component attribute sets tend to classify almost all instances as normal or as failures. This is most clearly the case for the IDC classifier and the file system classifier exhibits similar behaviour. These classifiers only perform better than a random predictor, which is indicated by a Kappa statistic that is greater than zero, when they simply classify most instances as normal ones. In problem domains in which instances of the outlier classes are rarer than their normal counterparts, classifying all instances as normal is typically a good guess. However, this does not lead to high accuracy in practice.

The attribute set for the condor scheduler process leads to more accurate failure detection. We hypothesised that this might be due to a higher degree of variety in the types of metrics, which may consequently provide a better basis for learning to discriminate between normal and faulty operation. However, the best accuracy so far has been achieved for a much more homogeneous set of metrics for the Tomcat class loading subsystem. When we inspect its instance data manually, we find that a high number of loaded classes appears to be particularly well-correlated with failures that affect the Tomcat JVM. Nevertheless, its accuracy does not fulfil our requirements on a mechanism for highly accurate failure detection. An overview of the accuracy of the per-component classifiers in the ROC coordinate space is given in Figure 7.7.

It appears that per-component attribute sets do not consistently lead to highly accurate classifiers. Instead, this fragmentation approach provides for a wide range of accuracy levels. Attempting to learn to detect failures by considering metrics of individual components in isolation may not be a promising

approach, given the interdependent nature of many of the components that we consider.

7.4.4 Top-X Classifiers

The final set of classifiers is based on subsets whose constituent attributes are selected by an automated mechanism that ranks attributes based on their suitability to form good decision trees. The one-class classification algorithm uses the class probability estimates of an internal two-class classifier (a decision tree in our case) in the process of defining a hypersphere that surrounds the target classes. A highly accurate internal classifier model should enable the one-class classification algorithm to define a well-fitting hypersphere around the target class region so as to discriminate more accurately between normal and failure instances.

As we have mentioned in Section 7.2.5, we use the C4.5 decision tree algorithm as the internal classifier. At each step this algorithm selects the most promising attribute from the remaining ones for building a decision tree of minimal height. The criteria on which the algorithm bases its selection is the information gain or gain ratio value of the attributes. Attributes that score higher values are preferred over ones with lower gain values. Attributes with higher information gain value separate a greater number of the remaining instances into one of two classes and thereby contribute a greater amount of information. We have used the WEKA workbench to rank attributes according to their information gain value[5].

With this approach, we can easily create differently-sized attribute sets. The smallest set is the singleton set that consists solely of the highest-ranked attribute. We then create larger sets by adding additional attributes in the order of their ranking, so that the resulting attribute sets are made up of attributes with decreasing ranking. The next attribute set of size two consists of the singleton and adds the next highly ranked one. In this manner, we create attribute sets consisting of the "best" single attribute, the "best" two, three, four, ten, twenty and forty attributes. Even if several attributes have the same information gain and consequently the same ranking, we simply select them in the order in which the ranker has output them. Each larger attribute set contains the smaller ones as its subsets.

Table 7.5: Number of unique components per attribute set.

	A	**B**
Top-1	1	1
Top-2	2	2
Top-3	3	3
Top-4	4	4
Top-10	9	9
Top-20	16	12
Top-40	29	16

We have created these differently-sized subsets based on attribute sets A and B. These serve as the initial base sets. The former constitutes the entire attribute set resulting from the monitoring data exclusive of some distracting attributes (cf. Section 7.4.1). We have chosen this attribute set as it provides

[5] We chose information gain over gain ratio as it led to more accurate decision trees in our experiments.

a wide range of different metrics to choose from, even though the accuracy of its resulting classifiers has turned out to be rather weak. Attribute set B only contains attributes that represent resource usage metrics. We have chosen it as the second base set because it provided for more accurate classifiers than set A in the previous experiments (cf. Section 7.4.2). The choice of these two sets allows us to compare the accuracy of the resulting classifiers in order to determine whether the composition of the initial attribute set has an impact on the effectiveness of the resulting subsets. We refer to the resulting attribute sets either as top-1, top-2 and so on or as A1, A2, B1, B2, etc. Table 7.5 lists the number of unique components whose metrics are represented in each of the subsets.

There are three key questions that we address with the help of these attribute sets. First, we find out whether automatically selected subsets based on a ranking of their information gain values constitute a good fragmentation approach that leads to classifiers with high detection accuracy. This is the same question that we have considered for the previous fragmentation strategies. In addition, we want to find out whether a larger initial attribute set provides an advantage when forming attribute subsets. That is, we want to know whether a base set that contains a larger variety of metrics can lead to an increased accuracy of the resulting classifiers. Third, we want to determine the impact that the number of attributes in a subset has on the accuracy of the resulting classifier.

7.4.4.1 Question 1: Highly Accurate Classifiers

We consider a classifier to be highly accurate, if it can detect at least 90% of failure instances and does not mis-classify more than 3% of normal instances as failures. In Figure 7.9 we have plotted the accuracy of the most accurate classifiers resulting from this fragmentation strategy in the ROC coordinate space. The figure shows which classifiers lie within the highly accurate quadrant and also shows a few of the classifiers in the immediate neighbourhood. Out of all the classifiers, A2 and A3 achieve the highest failure detection rate with a TPR of 96.2%. However, their FPR is at the limit for what we consider acceptable. A2 and A3 will mis-classify about three in about 100 normal instances, which translates to 11 false alarms per hour.

For failure detection, we generally assume that a higher TPR is preferable to the lowest possible FPR. However, in this case the performance of a few alternative classifiers is quite close to that of A2 and A3. These alternatives maintain a fairly high TPR, which is sufficient to meet our requirements for highly accurate failure detection, while at the same time achieving much lower false positive rates than A2 and A3 do. B20 has a TPR of 91.8% at an FPR of only 0.5%. B10 achieves the same FPR at a slightly lower TPR of 90.08%. As such it still meets our criteria for highly accurate failure detection. A4 has a considerably lower FPR at just 0.04%, which would translate to only 1 false alarm about every seven hours. Its TPR of 91.6% is only marginally lower than that of B20.

The peak accuracy of the five highly accurate classifiers is summarised in Table 7.6 along with the 95% confidence intervals around the observed mean TPR and FPR. The training and processing times for the five highly accurate top-x classifiers are shown in Table 7.7.

It is difficult to choose a clear winner from these results. Not considering classifiers A2 and A3, there is essentially a draw between A4 and B20 as the most highly accurate classifiers. The former

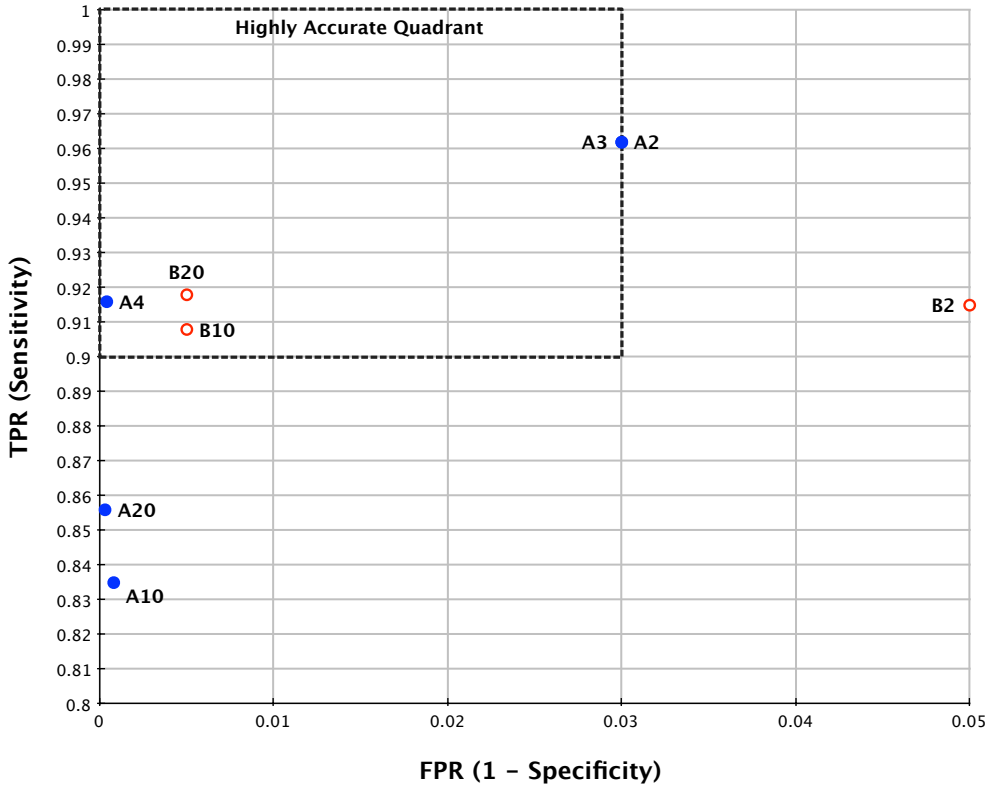


Figure 7.9: The top-x classifiers whose performance positions them within the highly accurate quadrant in the ROC coordinate space and in its immediate neighbourhood.

Table 7.6: Overview of the TPR and FPR of the highly accurate top-x classifiers along with corresponding 95% confidence intervals.

Set	Kappa	FPR	95% CI	TPR	95% CI
A2, 3	0.934	0.03	$(0.008, 0.052)$	0.962	$(0.951, 0.973)$
A4	0.935	0.0004	$(-0.0003, 0.001)$	0.916	$(0.894, 0.938)$
B10	0.924	0.005	$(-0.005, 0.014)$	0.908	$(0.889, 0.928)$
B20	0.932	0.005	$(-0.005, 0.014)$	0.918	$(0.903, 0.932)$

Table 7.7: Training and testing times in seconds for the highly accurate top-x classifiers.

	Training	Testing
A2	0.0250	0.0002
A3	0.0306	0.0001
A4	0.0396	0.0002
B10	0.0725	0.0002
B20	0.1468	0.0003

achieves near-perfect FPR and maintains an only marginally lower TPR than B20, which in turn misclassifies one normal instance out of every 200 as a failure (two false alarms per hour). A4 and B20 represent very similar alternatives.

With regards to the first question, whether or not this fragmentation strategy leads to classifiers that can support highly accurate failure detection, we can give a positive answer. We find a total of five classifiers that fulfil our requirements and that look promising for provision of failure detection in practice. These highly accurate classifiers are fairly close to each other in performance and offer different trade-offs between sensitivity and specificity. Which of these is preferable in a given application will probably have to be determined based on the cost of missed failures and false alarms as they exist for the deployment of a given service composition.

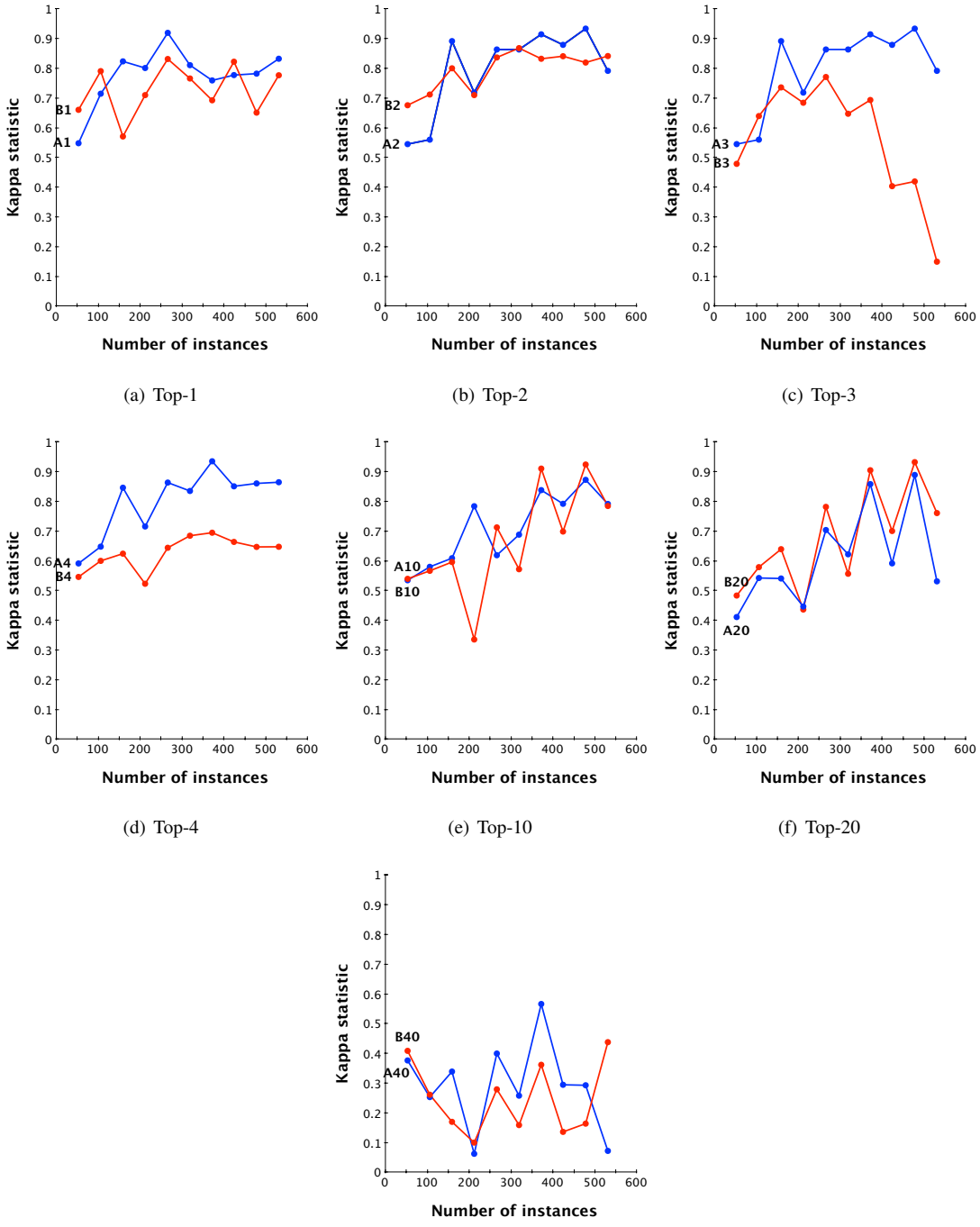
7.4.4.2 Question 2: Best Attribute Base Set

Attribute set A represents a much broader set of metrics consisting of over 500 attributes. Attribute set B focuses on resource usage metrics and is made up out of about 80 attributes. Given our results, it is difficult to decide conclusively which attribute set should be preferred for this fragmentation strategy.

The learning curves for all top-x classifiers are shown in Figure 7.10. The curves do not provide much useful information, but they reveal the following. The curves for both sets seem to indicate that higher accuracy could be achieved with the provision of additional training instances. However, this trend is not a clear-cut one as the Kappa statistic oscillates somewhat as the training set size increases. Even though the curves are quite similar to each other, classifiers based on attribute set A score slightly higher values for their Kappa statistic than those based on set B.

Next, we examine the composition of the attribute sets underlying the accurate classifiers in order to see whether they provide any clues about the source of their effectiveness and point to differences between the two base sets. Classifier A4 is based on four attributes that measure the physical memory used by one of the Condor daemon processes, the uptime of another such daemon, the uptime of the Monere agent process and the amount of used memory in one of the Tomcat JVM's memory pools (the Eden memory space). While uptime is intuitively a good indicator of failures, the distribution of their chosen attributes does not reveal their utility for this task. High uptime values for the Condor daemon are well-correlated with failures. The uptime values separate the instance space fairly well, but the way they do this is counter-intuitive. Furthermore, a system operator would probably not have chosen to focus her attention on the uptime of one of the Condor daemon processes in order to detect problems that affect the Polymorph service composition. There are other more obvious targets, such as metrics of the BPEL engine, SOAP runtime and application server. In contrast, the utility of the amount of physical memory used by the Condor daemon is easier to interpret. Very high as well as very low values are correlated with failures. As such, this attribute seems to serve as an indicator of the throughput in the system. The combination of these attributes leads to an accurate classifier.

The classifier based on attribute set B2 does not qualify as a highly accurate one. It consists of the free physical memory used by the Tomcat JVM and the free disk space available on the local file system. The distribution of its attributes is as expected (i.e. low values correspond to failures). A2,



(a) Top-1 (b) Top-2 (c) Top-3

(d) Top-4 (e) Top-10 (f) Top-20

(g) Top-40

Figure 7.10: Learning curves for the top-x classifiers based on attribute sets A and B.

which is based on the physical memory used by one of the Condor daemons and the uptime of another such daemon process, does make it into the highly accurate quadrant, even though the manner in which uptime separates the instance space, as we just explained, does not make sense. An expert might have preferred the attributes of set B2 over those that are in A2.

A comparison of the composition of attribute sets B20 and A20 does not reveal any more insight into why B20 is one of the best classifiers while A20 ends up with a considerably lower TPR of 85.6%. A20 draws its attributes from among availability metrics and several resource usage metrics such as JVM garbage collection, file system space, CPU load and the amount of virtual memory used by one of the Condor daemon processes. A20 represents 16 unique components and B20 represents 12. Out of these, eight components match across both sets. The additional components in set A20 include subsystems of the Tomcat JVM, Condor daemons, the network interface and the local file system. B20 is comprised of resource usage metrics about the CPU and Tomcat JVM. No considerable difference is apparent between these two sets. B20 is comprised of a set of metrics that would appear to form a good selection from the viewpoint of a human expert. However, the question remains why similarly good accuracy can be achieved with much smaller attribute sets (e.g., A4) and why classifiers based on A20 achieve a considerably lower TPR in spite of a relatively similar composition of the attribute set.

This brief examination of the composition of the attribute subsets does not reveal any insights that explain the high accuracy of the top-performing sets. In general we have found that well-performing attribute sets are not necessarily composed of metrics that a system operator might have chosen based on her experience and intuition[6] and the way the values separate the instance space can be unexpected. However, we need to bear in mind that one-class classifiers are generally opaque mechanisms that do no provide much explanatory information about the resulting classifications.

Overall, there appear more reasons to choose attribute set A as a base set for this fragmentation strategy, even though the performance of the highly accurate classifiers resulting from the two sets is quite similar. First of all, attribute set A produces three highly accurate classifiers compared with just two from set B. Second classifier A4 achieves one of the lowest FPRs while its TPR is still very close to the best classifier that set B has to offer (B20). Third, set A results in highly accurate classifiers based on a smaller number of attributes, which makes training and then classification at runtime more tractable. Fourth, if we consider the values of the accuracy measures in Table 7.8, it appears that attribute set A leads to more robust performance. Its accuracy degrades more slowly and steadily with an increasing number of attributes. The value of a larger initial set may be that it better supports unsupervised attribute selection mechanisms to select effective subsets. Confirmation of this hypothesis would require additional experimentation.

7.4.4.3 Question 3: Impact of Attribute Set Size

Finally, we examine the impact the number of attributes has on the accuracy of the resulting classifiers. Table 7.8 lists the Kappa statistic, TPR, FPR and the correlation coefficients for the number of attributes

[6]The internal classifiers, which we do not analyse here, are similarly opaque in that their structure does not fit with an expert's intuition (the author's) on how to detect failures.

and these measures.

Table 7.8: The peak accuracy achieved by the classifiers for the various top-x sets based on attribute sets A and B. Also shows the correlation coefficients for the number of attributes and each accuracy measure.

A			**B**	
Attributes	**Kappa**		**Attributes**	**Kappa**
1	0.832		1	0.822
2	0.934		2	0.868
3	0.934		3	0.771
4	0.935		4	0.695
10	0.873		10	0.924
20	0.889		20	0.932
40	0.566		40	0.438
correlation	-0.866		correlation	-0.629
Attributes	**TPR**		**Attributes**	**TPR**
1	0.950		1	1
2	0.962		2	0.915
3	0.962		3	0.839
4	0.916		4	0.665
10	0.835		10	0.908
20	0.856		20	0.918
40	0.611		40	0.381
correlation	-0.963		correlation	-0.746
Attributes	**FPR**		**Attributes**	**FPR**
1	0.108		1	0.129
2	0.03		2	0.05
3	0.03		3	0.082
4	0.0004		4	0.024
10	0.0008		10	0.005
20	0.0003		20	0.005
40	0.079		40	0.0009
correlation	0.163		correlation	-0.637

For the subsets of attribute set A there is a strong negative correlation between the number of attributes and the achieved Kappa statistic and TPR. That is, a larger number of attributes is associated with classifiers of lower accuracy. This is true at the amount of training data that we have used and could change were a drastically greater number of training instances applied. This result is as expected given the Hughes Effect and helps us to quantify the subset sizes that lead to highly accurate failure detection.

The TPR improves slightly at A20, before degrading again considerably at A40. We find only a weak correlation between subset size and FPR. The FPR is at its lowest for A4, A10 and A20, but considerably higher for the other subset sizes. We would have expected the FPR to increase together with an increase in the number of attributes, but cannot find a clear trend in our results to confirm this.

For the subsets of attribute set B we observe a negative correlation for all three measures, albeit the correlation is weaker for the Kappa statistic and for TPR than what we have been able to observe for set A. This can be explained by the fact that B achieves its highest accuracies at B10 and B20. The improvement in accuracy at a subset size of 20 is more pronounced for set B. Attribute set B also seems to lead to less robust performance. It begins with a strong TPR at B1 that decreases rapidly, then recovers again to reach its peak at B20, before decreasing even more significantly for B40. The negative correlation between the number of attributes and the FPR of the resulting classifiers is surprising. The FPR actually improves with an increasing number of attributes. This is contrary to what we would have expected.

The results paint a clearer picture for attribute set A than they do for set B. For the former the highest accuracy is indeed recorded for relatively small sets of attributes, even if not for the singleton set which has a relatively high FPR. Here, a higher number of attributes leads to lower accuracy. However our observations of the performance of attribute set B, muddle this picture to some extent and leaves us with two unanswered questions. First, why is there a recovery of accuracy at subsets sizes of around 20? What about these sets leads to an improvement in accuracy? And second, why does the FPR for classifiers based on attribute set B decrease consistently with an increasing number of attributes. Unfortunately, we are not going to examine these questions more thoroughly as part of this thesis. Instead, we answer our third question by asserting that we find evidence of a negative impact of the number of attributes on the failure detection rate and thereby on an important aspect of a classifier's accuracy.

7.5 Threats to Validity

7.5.1 Conclusion Validity

Threats to the *conclusion validity* of an experiment may negatively impact the ability to reach valid conclusions. First, we consider whether we have been able to make a sufficiently large number of observations. As we have explained in Section 7.2.3, we have used 10-fold cross-validation to calculate the outcome measures and have repeated this process 10 times per 10% training set increment in order to obtain learning curves. This process results in 1,000 observations for each classifier and 100 observations per training subset size. We have found that increasing the number of repetitions by a factor of 10 changes the resulting values of the outcome measures only marginally, but is computationally much more expensive. Therefore, and given that the number of observations is large we have reason to believe that we have obtained reliable accuracy measures.

Second, we consider the amount of training and test data. We have used a total of almost 100 minutes of monitoring data, 70 minutes of which represent data about normal system operation and 30 minutes of failures. In the form of instance data, this translates into about 420 normal and 180 failure

instances. Again, this is not a small number. In cases where the learning curves indicated that additional training data may potentially lead to improved accuracy, we have explicitly stated so. However, in these cases we expect that exponentially more data would be required. This would have increased the time and memory requirements of our cross-validation procedure significantly. Using 100 minutes of data allows us to perform a larger number of repetitions and appears to be a reasonable amount of data to train and test our classifiers given that some were able to achieve high levels of accuracy.

All three of our accuracy measures, the Kappa statistic, TPR and FPR, are objective measures. We have identified the potential weakness of the Kappa statistic for our problem domain. It rewards classifiers, even ones we do not consider to be highly accurate, simply because they classify the majority of instances as belonging to the target class. This can lead to optimistic results. Therefore, we have taken TPR and FPR into consideration for determining the accuracy of a classifier.

The evaluation and comparison of the classifiers based on these measures is not readily performed with the help of statistical hypothesis tests. We do not compute a single figure that represents the accuracy of a classifier, but instead rely on the combination of TPR and FPR values. An approach to determine differences of statistical significance among the various classifiers such as an analysis of variance (ANOVA) is not applicable under these circumstances. We have explained that the Kappa statistic is not a sufficiently reliable alternative based on which to assess classifier accuracy. We could have resorted to the use of a multivariate analysis of variance. However, in order to assess accuracy within the context of the application of such classifiers for failure detection in practice, a more nuanced and argumentative approach is required for their assessment. We need to be able to reason about and take into account the trade-offs that the classifiers offer. Even following a less objective approach, we have been able to identify a set of highly accurate classifiers and have briefly reasoned about the trade-offs they offer.

Another factor that influences the conclusion validity of an experiment is the reliability with which the treatments have been applied. This governs our ability to compare the results of several experiments with each other. We have used the same procedure to obtain learning curves through repeated 10-fold cross-validations on training set sizes in 10% increments on all attribute sets as explained in Section 7.2.3. We have characterised each treatment with an overview of the approach that constitutes a particular fragmentation strategy and the composition of the resulting attribute subsets.

Another concern is that our experiment design should make use of appropriate randomisation in order to reduce the risk of measuring the effects of any confounding factors. During the 10-fold cross-validation, each fold is used once for testing and the remaining ones for training. Each fold is composed of randomly selected instances (though instances are chosen so as to form stratified folds). Furthermore, for each training set size (i.e. 10%, 20%, 30% and so on) we reshuffle all folds 10 times. This is how variety is introduced in our experiment design.

7.5.2 Internal Validity

The *internal validity* of an experiment concerns the degree to which the independent variable is actually responsible for the observed effect on the dependent variable. The internal validity of our experiments is maintained for the following reasons. First, the only element that we vary between the experiments is

the fragmentation strategy and thereby the resulting attribute sets that serve as the basis for learning the various classifiers. The monitoring data used for training and testing is the same in each case as is the procedure by which we train and evaluate the classifiers. Second, the one-class classification algorithm has been implemented by its creators [The University of Waikato, 2012], has been available for several years and appears to work reliably. Finally, the experiment does not involve human participants. Therefore, effects from motivation, maturation and other such issues do not apply. The experiment that we use in this chapter is a fairly closely controlled one.

7.5.3 Construct Validity

The *construct validity* of an experiment describes the extent to which the treatments and the outcome variables represent the real-world cause and effect under investigation. The treatment is the fragmentation strategy or more specifically the attribute sets that result from it. The real-world cause is a classifier model learned from a fragmentation of the feature space represented by the multi-variate monitoring data. The cause construct is an artificial object, but this does not distract from the close correspondence between treatment and construct.

The primary outcome measures we use are the TPR and FPR computed for a classifier. The real-world effect this represents is the accuracy with which a classifier can detect failures in terms of its sensitivity and specificity. As such, there is a good correspondence between the measures and the effect. Ten-fold cross-validation is known to produce realistic accuracy measures. Furthermore, we have imposed a set of constraints over the values of the measures in order to define what we consider to be highly accurate failure detection. We have explained the behaviour the attainment of these parameters leads to in practice. This has allowed us to identify and compare a set of highly accurate classifiers.

We have not explored the entire space of possible levels for the treatment. There are many other fragmentation strategies and attribute sets that could potentially reveal interesting insights and help to clarify some of the open questions. However, we point to further experiments that may be worthwhile to pursue in the final chapter.

7.5.4 External Validity

The *external validity* of an experiment describes how well its results generalise to a wider context. We use monitoring data from a single service composition and from a limited set of failures. As we have explained in Chapter 5, the Polymorph service composition has the key system characteristics we are interested in and is of moderate size and complexity. The failures are ones that have been observed in production use of the service composition and the conditions leading to them are reliably reproduced by our failure injection tool. Furthermore, we have confirmation that the failure symptoms are indeed detectable in the data through the results of our diagnosis experiments. Both the service composition and the failures are representative of the types of systems and failures we are targeting.

The accuracy of the outcome measures is based on the measurement procedure we apply. The ability of 10-fold cross-validation to produce realistic accuracy measures has been noted in [Witten and Frank, 2005, page 150]. The use of this method increases our confidence that the measured accuracy is close to what would result from application of the classifiers in the field.

Even though we have reason to assume that the conditions to which we have exposed the classifiers are close to what they would encounter in practice, we cannot know whether application in the field over long durations may uncover conditions that we have not been able to cover. We have attempted to counter this threat by incorporating several runs and modes of operation.

7.5.5 Repeatability

The final question we consider is whether or not our experiment is *repeatable* so as to enable independent verification of our results. We have described the experiment setup in detail. The WEKA workbench and its implementation of the algorithms are freely available and we have given details about how we have configured the algorithms. This should enable replication of our experiments with different sets of monitoring data.

7.6 Discussion

We set out to address two related questions about data-driven failure detection. First of all, we wanted to determine whether it was possible to implement a highly accurate failure detection mechanism based on applying one-class classification to multi-variate monitoring data. Second, we wanted to find out how to select attribute subsets from a large set of metrics so as to make learning such classifiers both tractable and highly accurate.

We were able to confirm that the feature space formed by the monitoring data of even a single host is too large to learn an accurate one-class classifier. Our first classifier, which consisted of attributes that represented all metrics collected for the components on host huey, achieved a TPR of less than 18% at an FPR of 1%. Next, we examined three different approaches to fragment the feature space, each of which resulted in several attribute subsets. The first of those was a fragmentation based on metric type. The idea behind this approach was that certain failures affect a particular aspect of the behaviour of the system, which in turn becomes visible through certain metrics. We examined the classifiers learned from resource usage and from performance metrics. The latter had very low accuracy and while the former was an improvement over the complete attribute set, with a TPR of 34% it was still far from qualifying as a highly accurate failure detection mechanism. From our results it appears that a fragmentation based on metric types is not an effective approach.

As the next fragmentation strategy, we examined the performance of four per-component classifiers, the idea being that a component forms a natural unit of failure detection in a service composition. Each of the four attribute sets in this case consisted solely of the metrics of its component. Together, the four selected components are affected by all six failures, but we have tested each classifier only on the failures that directly impact its component. Even though this is an intuitively appealing approach, the best accuracy we have been able to observe was for the Tomcat class loading system with a TPR of 77.5% and an FPR of 1.2%. This still fell short of what we consider to be sufficiently accurate for failure detection in practice. It is possible that classifiers are able to exploit the inter-dependencies that exist between some components and that learning to detect failures for components in isolation is not a promising approach for the types of failures that we consider.

The final fragmentation strategy that we have considered was to rank attributes based on their ability to form accurate decision trees and automatically create differently-sized subsets based on such a ranking. We used attribute sets A (all metrics) and B (resource usage metrics) as the initial base sets for this approach. In addition to determining the accuracy of this fragmentation strategy we also wanted to find out whether a larger or smaller initial set was preferable and examine the relationship between the number of attributes and the accuracy of the resulting classifiers. The attributes of this fragmentation strategy resulted in five highly accurate classifiers, each with a TPR greater than 90% and with FPRs that ranged from just 1 false alarm every seven hours to 11 false alarms per hour. These classifiers represent several alternatives that could be used to automate the detection of system-level failures.

While the automated generation of subsets evidently supports highly accurate failure detection, we have not been able to determine what about the composition of the attribute sets and the distribution of their values makes them effective. A surprising result is that the best-performing subset selections do not necessarily conform to what an expert might have selected. We were not able to explain why these attribute sets serve as good discriminators between monitoring data that represents normal operation and failures. We found that classification accuracy, in general, decreases with an increasing number of attributes. However, our results about the impact of the number of attributes leave a number of open questions, such as why there is an improvement in accuracy at a subset size of about 20 or why the FPR for set B decreases monotonically with an increasing attribute size contrary to our expectation. From our results it appears that a larger attribute set offers a better basis for this automated approach to fragmentation. It is possible that an increased variety of metrics to choose form, increases the likelihood of finding effective subsets.

Our approach to data-driven detection and our evaluation of it are subject to a number of limitations. We motivated our work, at least in part, with the goal of drastically reduced failure detection times. However, we were not able to measure this effect directly. The implementation of the one-class classification algorithm that we have used, requires all failure instances to be labelled as “outlier”. This then prevents us from establishing which individual failures were mis-classified or classified correctly. At low levels of TPR we do not know which of the six failures are detected. Furthermore, we cannot establish what proportion of the instances that belong to a particular failure are classified correctly or how early after activation of a failure it is detected (i.e. which of its instances is the first one to be detected).

However, at a TPR of at least 90% there is a strong likelihood that a classifier detects all failures, that it does so reliably (i.e. correctly classifies most of the instances belonging to a failure) and early after the occurrence of a failure. A highly accurate classifier will detect one of the first instances of monitoring data about a failure (as it detects a very large proportion of all failure instances). If it were to detect the first failure instance for a failure, then detection should occur within 10 seconds of the activation of that failure. Otherwise, detection may take in the range of a few tens of seconds. Detection times are also influenced by how long it takes the classifier to process instances. We have measured the processing times for most classifiers on the largest test sets that we had available and found that on average processing about 60 instances took several hundred micro-seconds. Therefore, we can conclude

that highly accurate classifiers afford relatively fast failure detection that can be made to operate within seconds.

Another limitation is that we have evaluated the various classifiers on failures of relatively long durations. This has provided us with more challenging tests sets for the classifiers, but means that we do not know how well our approach works with failures of short durations. A repetition of the experiments with a different set of failure instances would be necessary to clarify this question.

Our analysis to determine what makes certain attribute sets effective has not been successful. Therefore, one further limitation of our approach is the lack of explanatory information about why a failure has been detected. Neither the internal classifier not the attributes of the highly accurate classifiers reveal much useful information about how failure detection is performed. A consequence of this is that our approach does not assist in the localisation of failures. However, our goal was primarily to achieve highly accurate failure detection and as such localisation would have simply been an added bonus.

In summary, we have been able to confirm that applying one-class classification to multi-variate monitoring data can indeed form a feasible approach to automate the detection of system-level failures in a highly accurate manner. We have found an effective fragmentation strategy to address the feature space problem. This strategy is a largely automated one and requires little manual effort. Furthermore, we have found that automatically generated attribute subsets appear to achieve substantially higher accuracy than those selected based on an expert's intuition and experience. Finally, applying this fragmentation strategy to a large initial set of attributes leads to highly accurate classifiers that are based on small numbers of metrics. Data-driven detection may operate in a somewhat opaque manner, but it does so fast and reliably.

Chapter 8

Conclusions

In this chapter we review the contributions we have been able to make in this thesis and evaluate them in terms of the new knowledge they create as well as by identifying the open questions that remain. We end the chapter with a few suggestions for further work.

The overall problem that we have addressed in this thesis is how to improve failure detection and diagnosis in the context of cross-domain, middleware-based service compositions. Middleware facilitates the development of complex applications as compositions of basic services and thereby the integration of widely distributed and heterogeneous resources. However, the resulting applications inhabit a complex operating environment that subjects them to numerous opportunities for failures. We have focused on system-level failures. These prevent an application from making progress, unless they are detected, their cause is identified and resolved. They originate in the middleware, the network or the operating system and their effects can propagate across system layers and even across administrative domains. Furthermore, system-level failures are usually due to operational faults that occur at runtime and often need to be handled at that stage. The manual detection of system-level failures is time-consuming, slow, costly and error-prone. Similarly, a diagnosis process that primarily relies on the experience and intuition of system operators does neither provide adequate support for an efficient identification of failure causes nor does it scale well with the increasing size and complexity of software systems. This leaves room to address the availability of service compositions by reducing the time of these two key activites that comprise the mean time to repair failures. Our contention was that the currently prevalent approach to detection and diagnosis could be improved on.

In order to do so, we have proposed and then examined the characteristics of a data-driven approach to the detection and diagnosis of system-level failures. This approach is based on the comprehensive, but non-intrusive, monitoring of the operation of a service composition and the discovery of its structure. The former by itself enables the automation of failure detection in a highly accurate manner. The data made available from both monitoring and dependency discovery supports system operators during diagnosis by allowing them to perform diagnosis in a more principled manner. It complements their skills with access to data about system operation and structure and as we have found enables them to achieve significantly higher success rates and shorter diagnosis times. We have found that such a data-driven approach represents a promising approach to reduce the MTTR of service compositions and can

thus complement existing approaches that attempt to maintain the availability of software systems by increasing their MTTF. In the course of our investigation of this data-driven approach, we have had the opportunity to make four contributions. Two of these contributions are of a primarily technical nature, while the other two contribute new knowledge.

8.1 Contributions

8.1.1 Comprehensive Non-Intrusive Monitoring

The first technical contribution is a framework for monitoring the operation of most of the components that comprise a middleware-based service composition in a non-intrusive manner that we have presented in Chapter 3. The framework integrates and correlates data that would otherwise remain scattered throughout the system and might not be available when needed by system operators in some cases. The framework creates and maintains a repository of historic records of measurements for numerous metrics that describe several aspects of the operation of a service composition.

The monitoring framework does not address any research questions by itself. Instead, it forms an important element of our data-driven approach and as such enables us to examine its characteristics. The monitoring framework supports the intrinsic evaluation of our approach by demonstrating that it is possible to extract relevant data from middleware-based service compositions without changing services and appliactions. Its implementation allows us to determine the feasibility of our approach.

Nevertheless, our work on the monitoring framework makes some contributions, given that no such approach to comprehensive and non-intrusive monitoring of a service composition at all its layers has been described in the literature. We have been able to quantify the performance overhead that is introduced by such an approach to online monitoring. We have measured the slowdown to a monitored service composition, determined the resource usage requirements of the monitoring agents and analysed and measured the communication overhead within and across administrative domains. The performance cost appears to be tolerable by modern servers and networks and even though it is not negligible, it is likely to be sufficiently modest to enable service providers to realise the benefits of data-driven detection and diagnosis in terms of reduced times for these two activities. Furthermore, our work on the framework demonstrates what measurement approaches are available to monitor most middleware-based systems non-intrusively. In particular, we have described the use of event tracing, interception of request messages and sampling. We have learned what types of metrics can be obtained through these approaches, such as availability metrics, information about exceptional events and application activity, resource usage and performance metrics. Finally, we have been able to describe the management information base that results from such a monitoring approach and argued how this information can support a data-driven approach.

8.1.2 Cross-Domain Structural Dependency Discovery

The second technical contribution is a process for the automated discovery of structural cross-domain dependency graphs that we have presented in Chapter 4. The process exploits deployment information that is commonly available in middleware-based systems along with information contained in service

composition artefacts. The resulting dependency graphs reveal most parts of the critical infrastructure of a service composition. The relationships they contain represent calling relationships among Web services within and across administrative domain boundaries as well as intra- and inter-host containment and other explicit dependencies onto and among the middleware and operating system components.

Our work on dependency discovery does not directly address any specific research questions. However, it enables us to examine what the impact on the diagnosis of system-level failures is when we supply system operators with structural information about large, cross-domain systems.

In addition to the above question, we have been able to gain an insight into the types of dependency graphs that can be created by exploiting deployment information as it is commonly reified in various forms by middleware. We have briefly evaluated the accuracy and completeness of the graphs our process discovers in Section 5.4. Our structural dependency graphs compare favourably with functional ones that are based on runtime observations. All components and relationships that are represented in our graphs do exist in the given deployment. However, the graphs also contain dependencies that may only become active rarely. Furthermore, our approach is not able to discover data dependencies between components. Such relationships cannot be revealed based on a static analysis of the sources of deployment information that are available in middleware-based systems. Finally, we were able to explain how the availability of such dependency graphs enables the efficient exchange of measurement data between service providers and their clients across domain boundaries by supplying clients with a record of their remote dependencies.

8.1.3 Data-Driven Failure Diagnosis

The third contribution is based on the results and insights that we have gained from an evidence-based evaluation of our approach to data-driven diagnosis.

The first question we set out to address was concerned with the feasibility of the kind of monitoring and dependency discovery necessary to make the data-driven approach possible. We have addressed this question with a performance analysis of the former and a qualitative review of the graphs resulting from the latter as described in the previous two subsections.

Next, we investigated whether and in how far data-driven diagnosis could improve the diagnosis of system-level failures. We examined its benefits with a controlled experiment that measured and compared the performance of 22 participants with the task of diagnosing a set of failures that we injected into a cross-domain service composition. The participants were split into two groups. The control group used a traditional diagnosis process supported by a standard tool set. The effect group used our prototype to gain access to historic and real-time data about system operation as well as information about system structure, which allowed them to engage in data-driven diagnosis.

We have quantified the effects of data-driven diagnosis in terms of success rates and diagnosis times. We found that participants using data-driven diagnosis were able to achieve significantly higher success rates than their counterparts who were using a standard approach. An operator who uses data-driven diagnosis is likely to conclude a greater number of failure diagnoses within a given amount of time than someone relying on a combination of ad-hoc measurements, experience and intuition.

Furthermore, we were able to confirm our expectation that data-driven diagnosis can support system operators to achieve significantly reduced diagnosis times. This is particularly true for failures whose cause lies in remote administrative domains. Our results have enabled us to determine the circumstances under which its benefits justify the application of data-driven diagnosis. We have explained how this depends on several factors, such as the cost of downtime as agreed in service-level agreements, the proportion of repair time that is generally expended on diagnosis and the degree to which an application integrates resources that are widely distributed. In general, we have found that service providers should be able to realise the benefits of data-driven diagnosis when overall failure repair times are dominated by diagnosis times, whenever the penalties incurred from downtime are substantial and for service compositions that routinely use resources in different administrative domains so that failures can propagate their effects widely across administrative boundaries.

The third key insight we gained was that dependency graphs represent a useful tool to guide system operators more quickly towards regions of the problem space that are worthwhile to investigate more closely. We have also learned that inaccurate or missing relationships in such graphs can be misleading and have a prominent negative impact on the effectiveness of the diagnosis process.

Overall, we performed and reported on the first evidence-based determination of the effects that such an approach has on the diagnosis of system-level failures, quantifying its benefits, revealing its strengths and weaknesses and identifying the circumstances which justify its application.

8.1.4 Data-Driven Failure Detection

The final contribution consists of our evaluation of automating the detection of system-level failures by applying machine learning techniques to multi-variate monitoring data. In particular, we evaluate the suitability of an outlier detection technique, one-class classification, to learn to detect failures from multi-variate monitoring data about system operation.

We set out to address two closely related questions. The first one was whether our chosen outlier detection technique, one-class classification, could learn an accurate model of normal system operation based on multi-variate monitoring data and then use this model to accurately detect data that is indicative of failures. The second question was concerned with how to solve the intractability of learning an accurate model from the large feature space that is the result of our multi-variate monitoring data. We have defined what we understand by highly effective failure detection with regard to the failure coverage (true positive rate) that a classifier should achieve and the number of false alarms that it would confront system operators with (false positive rate). In order to find out how to select effective subsets of metrics, we have applied different fragmentation strategies to obtain smaller subsets and trained and tested classifiers on close to 100 minutes of monitoring data about normal operation and failures.

We were able to create a total of five highly accurate classifiers, each offering slightly different trade-offs between failure detection coverage and false alarm rates. Each of the five classifiers achieves a true positive rate of more than 90% and their false alarm rates range from one false alarm every seven hours to 11 per hour. We found that fragmentation strategies that rely on an expert's judgement to select attribute subsets did not support highly accurate failure detection. Instead, the best approach was an

automated one that ranks attributes by their suitability for generating accurate decision trees and then creates differently-sized subsets ordered by the ranking of the attributes. Small subsets of attributes (two, three or four attributes) are sufficient to train highly accurate classifiers and generally smaller numbers of attributes are well-correlated with higher levels of accuracy. However, our results leave a few open questions in this regard. We have not been able to determine what about the composition of the five automatically generated subsets leads to a high degree of accuracy in the resulting classifiers. Nevertheless, we have found that the automated approach we tested enables data-driven failure detection that detects all failures, does so reliably and in a timely manner.

8.2 Open Questions and Future Work

We have been able to answer our initial questions about the data-driven approach and have gained some interesting insights. However, our work leaves several worthwhile questions open and a few ideas unexplored.

Our results on data-driven diagnosis did not reveal any significant impact of the experience level of an operator on diagnosis times. As we noted, we did not have a sufficiently large number of observations to conclusively address this question. A partial replication of our controlled experiment might provide further insight into this. A new set of participants, all using our prototype for further failure diagnoses, could be divided into possibly three blocks according to their experience (e.g., low, medium and high). Such a blocked experiment design could either confirm our results of a lack of impact or otherwise be able to better quantify the effect of experience on diagnosis times.

While the benefits of data-driven diagnosis when applied to system-level failures is by no means insignificant, the effects are not as dramatic as achieving a halving of diagnosis times. This may, in part, be the result of the relatively short times that we were able to allocate per failure, but it may instead point to another insight about our approach.

The availability of measurements about system operation is important and necessary to ensure the manageability of middleware-based software systems of non-trivial size and complexity. Whoever has experience with the maintenance of distributed software systems would probably intuitively agree with this statement. However, we recognise that using an approach such as ours, a system operator is still faced with a large volume of data. Mechanisms to obtain rich repositories of such data may only be a building block, even if a necessary one, for providing higher-value added functionality.

As our work on automated failure detection demonstrates, a promising approach to further improve the handling of unanticipated system-level failures may be to investigate automated mechanisms that can detect anomalies and then perform further analyses to identify other areas of interest within a large volume of data and thereby effectively reduce the problem space on behalf of a system operator. This leads us to two suggestions for future work to build on the efforts presented in this thesis.

8.2.1 Benchmark of Failure Detection Mechanisms

It would be interesting to expose a variety of machine learning techniques to a kind of benchmark in order to determine their suitability for and compare their performance with failure detection based on

multi-variate monitoring data. At a minimum, such a benchmark should test the following characteristics.

- **Training sensitivity.** How much training data does a technique require in order to achieve its peak accuracy?
- **Workload sensitivity.** How robust is a mechanism against changes in workload that were unseen during training?
- **Prediction ability and accuracy.** As we have explained, the monitoring data our framework produces forms a multi-variate time-series. Some techniques may be able to use such data in order to predict the occurrence of failures. It would be interesting to find out how far into the future it is possible for a mechanism to predict failures and determine the relationship between the size of the prediction window and the resulting accuracy.
- **Time to detection.** Another characteristic based on which to compare different techniques is the speed with which it is able to detect failures.

Such a benchmark, performed on a set of machine learning techniques, could lead to interesting insights about the characteristics that make a technique particularly effective at the task of failure detection and prediction and could reveal more knowledge about the performance with which we can automate failure detection (and possibly prediction) based on the processing of multi-variate data about system operation. However, a great deal of realistic monitoring data would be required in order to systematically examine all these questions.

8.2.2 Spatial Scan Statistics for Failure Root Cause Localisation

Our approach to failure detection does not provide much information that would assist a system operator with the localisation of system-level failures. Per-component classifiers could remedy this situation by pointing to the component in which a failure has been detected. However, as we have found, learning to recognise failures in a component in isolation from other components does not appear to provide high accuracy; at least not for the mechanism we have evaluated. Failures tend to cascade and often affect more than just a single component.

It might be possible to provide for localisation and improve the accuracy of per-component classifiers with the help of our dependency graphs and by borrowing concepts from spatial scan statistics. Spatial scan statistics [Kulldorff, 1997] are commonly used for geographic disease detection. Without going into too much detail, the basic idea is to examine the locations of occurrences of a phenomenon (e.g., reports of patients who have become infected with a particular disease) and then perform significance tests to determine whether there is a region with significantly high or low rates of occurrences when compared with other regions. A comparatively higher rate might indicate the outbreak of an epidemic and probably warrants raising an alarm.

In our context, we could create a suitable "coordinate system" by using the dependency graphs our approach discovers. We could use the graphs to define a local neighbourhood of components or regions around each component based on its set of dependencies and dependents. This would allow us to apply

spatial scan statistics to the alarms raised by a set of per-component classifiers. The alerts raised by each per-component classifier would not immediately be used to notify a system operator and declare a failure, but would in the first instance be processed by spatial scan statistics to take recent observations in a region into account and compare it with the situation in other regions. In this way, it might be possible to improve the overall true positive rate by giving consideration to the output of stronger detectors alongside that of weaker ones. At the same time, this could reduce the false positive rate as alarms would only be raised, if the alarm rate from several individual components in a region becomes significantly higher than in the rest of the system. We cannot anticipate whether or how well such an approach might work in practice, but it may be a promising way to achieve highly accurate failure detection and localisation in large systems where data about system operation and system structure are available.

This concludes the description of our work on data-driven failure management in middleware-based service compositions. We have been able to explore a number of interesting questions and hope that the results and insights presented in this thesis may be of use to the community of researchers and practitioners as another small building block on the path toward distributed software systems that routinely achieve high levels of availability.

Appendix A

Monere Prototype

In this appendix we provide a more detailed overview of the *Monere* prototype framework than was possible in Chapter 5.

A.1 Overview

The Monere prototype is based on an early version of the open-source RHQ enterprise management system [Red Hat, Inc., 2012]. RHQ provides a set of core services for systems management, such as an abstract inventory model and the discovery of some hardware and software components. RHQ can only represent simple containment hierarchies among components on a single host. Monere modifies and extends RHQ in order to make it suitable for cross-domain service compositions and implements the features that we have discussed in Chapters 3 and 4. Monere adds additional component types, modifies the discovery and measurement behaviour of existing ones and defines additional types of metrics. It also extends the discovery capabilities of RHQ by integrating additional types of dependencies (i.e. among middleware components and services) that span host as well as administrative domain boundaries. Monere also contributes a mechanism to efficiently exchange measurements across domain boundaries and provides a web-based user interface for use by system operators during diagnosis of system-level failures.

The three main components of Monere are the Monere server, agents and the Monere Information Service (MIS). They are shown in Figure A.1 as deployed across two administrative domains in our testbed. Each domain hosts a central Monere server, which communicates with the set of Monere agents deployed on the hosts in its domain. The server obtains component discovery results and measurements from these agents and maintains an inventory for all hosts in its domain. The server persists the inventory in a database and makes this data available to end users via a web-based user interface. The Monere server is also responsible for processing and merging the individual sub-graphs discovered by its agents. In case the server has not received any measurements from an agent within a certain time frame and is unable to reach this agent, it will record the agent along with the components it has discovered as temporarily unavailable.

A single Monere agent is deployed on each host where it implements the local discovery process. The agent, which runs within its own Java process, is responsible for the discovery of any components on its host and for the measurement of their metrics. An agent periodically executes the discovery

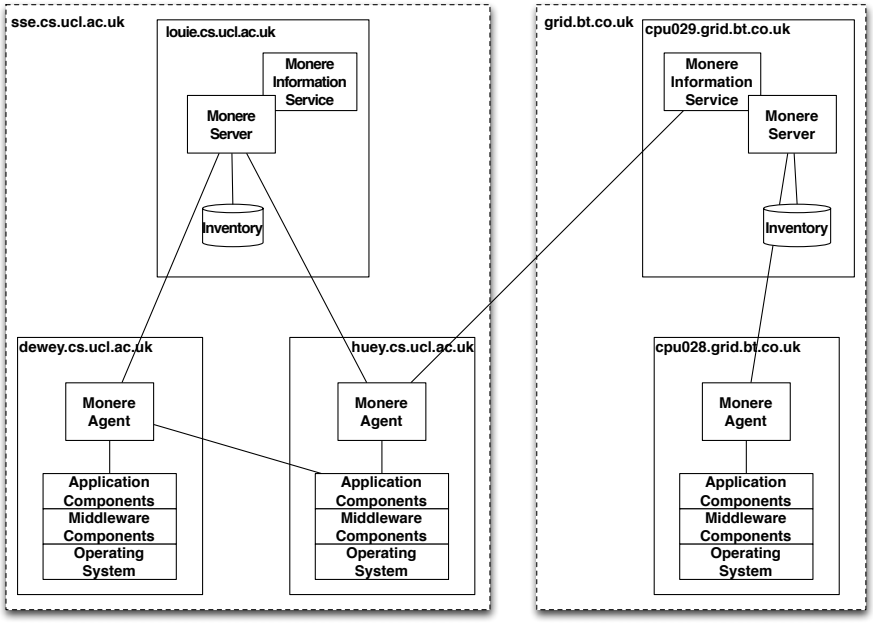


Figure A.1: The key components of the Monere prototype as it is deployed on our cross-domain testbed.

process in order to detect any changes in the local deployment. Measurement collection is driven by user-defined collection intervals for each metric. The smallest supported interval at which an agent takes measurements is one second. Suitable collection intervals for our case study have been determined experimentally and range from five to 300 seconds. An agent stores all measurements for the components it has discovered in a measurement report and then checks this report every 30 seconds to determine whether it contains new data that needs to be sent to the server.

An agent is essentially a runtime system for a set of plugins and manages communication with the Monere server. Once an agent has been deployed on a host, it registers with the Monere server in its domain and downloads any needed functionality in the form of plugins. A plugin represents a particular component type and encapsulates the functionality for its discovery and the measurement of corresponding metrics. Plugins consist of a descriptive part in XML and a Java implementation. The former specifies information about the represented component type and defines a set of metrics to be collected from instances of this component type along with data type definitions and user-configurable collection intervals. The implementation of resource discovery and measurement collection is performed by the corresponding class files referenced in the plugin descriptor.

The MIS implements our approach to the exchange of measurements between service providers and their clients across domain boundaries (cf. Section 4.5). It addresses the requirement of providing visibility of dependencies on components in remote administrative domains and sharing information about their state. A service provider can make metrics about its published services available to clients by publishing them as RSS feeds at the MIS responsible for its domain. An agent that has discovered a dependency from one or more of its local components onto a Web service in another administrative

domain, can subscribe to the corresponding RSS feed at the MIS of that domain by using the endpoint URL at which the service is invoked as a key to identify. In this way, service providers do not have to maintain up-to-date lists of clients that need to be notified. The MIS restricts visibility to the level of published Web services and provides no further insight about the underlying infrastructure. It safeguards the sensitivity of implementation details, while providing data that can help to determine problems with remote services.

A.2 Measurement Techniques

Monere uses various techniques to achieve a non-intrusive implementation of the three measurement approaches that we have discussed in Section 3.4. The first measurement approach is event tracing. Its implementation in Monere is based on two techniques. One consists of parsing the log files of middleware components and of log files maintained by the OS in order to extract relevant information. Monere parses new lines and determines whether they contain information that may be of interest based on the severity of a logged message (i.e. error and warning messages). This technique does require that log files adhere to common formats, which in most cases is a matter of configuration, but does not necessitate any intrusive changes in order to work. Another technique for event tracing in Monere is to subscribe to notifications of state changes for BPEL processes via the administrative interface of a BPEL runtime, such as that offered by the ActiveBPEL engine [Active Endpoints, 2012b].

Interception is another measurement approach. One way in which Monere implements this approach non-intrusively is by insertion of a custom message handler within the message handling chain of a SOAP runtime. For example, in the Axis SOAP runtime [The Apache Software Foundation, 2012a] both outgoing and incoming SOAP messages pass through a sequence of message handlers, each of which can perform some action based on the information in the message header or body. A message handler might count the number of messages addressed to a particular endpoint or execute some custom transformations or formatting. Adding a custom message handler to such a chain of handlers is done by modifying a configuration file made available by the SOAP runtime. Monere's message handler is thus able to intercept all SOAP messages that are exchanged between Web services and BPEL processes. It reads and writes to the message headers in order to obtain and record measurements, such as for example request latencies and execution times.

We referred to the third measurement approach as sampling. There are several different techniques to implement sampling in a non-intrusive manner. Some components written in Java make use of the Java Management Extensions (JMX) [Sun Microsystems, 2006]. JMX defines interfaces that allow objects executing within a JVM to be represented by so-called managed beans (MBeans), written by the developer of an application. These MBeans can contain configuration information for an application as well as act as probes that expose arbitrary measurement data, such as its performance and resource usage. At runtime, these MBeans can discover and register with a so-called MBean or JMX server. Interested clients (i.e. a Monere agent) can then connect to an JMX server, query its list of MBeans and retrieve data from ones that represent relevant components. Monere uses this approach to obtain numerous metrics about the operation of application servers, such as the amount of heap space or the number

of Java threads used. Another implementation technique for sampling used in Monere is to issue HTTP HEAD requests to components that accept HTTP requests to check their availability. For some of the metrics about the operation of the OS and its components, Monere uses a set of native libraries that take measurements of many OS metrics [Hyperic, 2012]. Another non-intrusive instrumentation technique is to query data in the OS process table. And finally, as a last resort, Monere can execute shell commands and parse their output.

We identified the need to correlate application activity with measurements taken at the other layers. Monere implements a simple correlation mechanism based on timestamps. For each application activity Monere records its start and end time. Similarly, agents record timestamps for any other measurements they take. Synchronisation among agents and servers is performed using the Network Time Protocol (NTP) [Mills et al., 2010], which achieves synchronisation of clocks across the Internet to within a few milliseconds of each other. Given that the smallest collection interval supported by Monere is one second, this degree of accuracy is sufficient for our purposes. In order to correlate application activity with lower-level measurements, Monere calculates a time range of a few seconds on either side of the occurrence of an activity and then includes all measurements that fall within this time range as being correlated with it. While correlation based on timestamps cannot guarantee to establish causal relationships, it can afford insight into how different parts of the system reacted at the time the application performed a particular activity.

A.3 Dependency Discovery

In this section we discuss the implementation of the static analysis of service-level artefacts. We have given a procedural overview of the entire discovery process in Section 4.4. There we have explained how the components in a given deployment and their dependencies are discovered in a piecemeal manner through a series of procedures. We listed the deployment information that these procedures can make use of in middleware-based systems. The service-level static analysis is one of these procedures. It complements the information obtained by the other procedures with calling dependencies among services.

As we mentioned before, service composition languages typically provide facilities to specify the set of services in a composition. The implementation of the static analysis of service-level artefacts in our prototype caters for service compositions that are expressed as BPEL workflows. Accordingly, it exploits several language constructs of BPEL and WSDL in order to identify the orchestrated Web services. The relevant constructs are contained in Listings A.1 and A.2, which show excerpts of a BPEL process specification and of the WSDL description of one of its partner services.

Like most XML-based documents, BPEL processes define a set of namespaces and corresponding prefixes in a preamble as is shown in lines 3 to 8 in Listing A.1. The preamble declares the target namespace (line 3) that all elements of this document belong to. This is similar to a Java package declaration. It is followed by declarations of prefixes (usually in the form of a short string, such as “plot”) and the namespaces (usually given in the form of a string that looks similar to a URI) they refer to. Starting on line 11 there are two import elements. These are optional elements that allow a BPEL process to specify resources that define some of the namespaces used in the process. Again, this can be

compared to Java import statements. The location of an import element can be a relative URI pointing to a local file or an absolute one in the form of a URL. Each BPEL process is itself represented by a WSDL interface that defines the messages it can accept as input and that it returns as a result.

Listing A.1: Snippets of a BPEL process specification.

```
1 <?xml version="1.0" ?>
2 <process name="visualizer"
3      targetNamespace="http://cs.ucl.ac.uk/omii/bpel/visualizer"
4      xmlns:cml="http://www.xml_cml.org/schema/cml2/core"
5      xmlns:jsdl="http://www.ggf.org/namespaces/2004/12/jsdl-1.1.xsd"
6      xmlns:tns="http://cs.ucl.ac.uk/omii/bpel/visualizer"
7      xmlns:plot="http://cs.ucl.ac.uk/omii/bpel/plotutil"
8      xmlns="docs.oasis-open.org/wsbpel/2.0/process/executable"
9 >
10
11     <import importType="http://schemas.xmlsoap.org.wsdl/"
12 location="wsdl/visualizerService.wsdl"
13 namespace="http://cs.ucl.ac.uk/omii/bpel/visualizer"/>
14
15     <import importType="http://schemas.xmlsoap.org.wsdl/"
16 location="http://huey.cs.ucl.ac.uk:18080/axis/services/plotutil.wsdl"
17 namespace="http://cs.ucl.ac.uk/omii/bpel/plotutil"/>
18
19     <parterLinks>
20         <partnerLink name="visualizerClient" partnerLinkType="tns:
               VisualizerPLinkType" myRole="VisualizerPLinkProvider" />
21         <partnerLink name="plotutilPartner" partnerLinkType="plot:plotutilPortLink"
                partnerRole="plotutilPortProvider" />
22     </partnerLinks>
23
24     <variables>
25         ...
26     </variables>
27
28     <sequence name="" joinCondition="" suppressJoinFailure="">
29         ...
30         <invoke partnerLink="plotutilPartner" portType="plot:plotutilPort"
               operation="storeResults" inputVariable="storeResultsRequestMessageVar"
               outputVariable="storeResultsResponseMessageVar"  name="storeResults"
               joinCondition="" suppressJoinFailure="">
31         </invoke>...
32         ...
33     </sequence>
34 </process>
```

In lines 19 to 22 the BPEL process defines a set of partnerLinks. Each partnerLink specifies one of the partner services that the BPEL process interacts with. It lists the roles the two parties need to assume in their interaction and hence which operations and message types their interfaces need to support for

such an interaction. For example, the partner service can act as a client while the BPEL process acts as a provider. Or similarly, one party can act as the seller in an interaction and the other one as the buyer. The partnerLlink assigns a name that is to be used internally to the BPEL process in order to refer to the partner service. It also references a partnerLinkType (PLT) that is defined by a WSDL document at a given namespace that is indicated by its corresponding prefix. The definition of the PLT for the plotutilPartner is shown in Listing A.2 in lines 38 to 42. The PLT makes a reference to a portType, which is in turn defined as the type of a binding element (lines 18 to 30). A binding element defines the transport binding and style of SOAP messages to be used for each of the operations offered by this service. The binding itself is then referenced from a matching service element. The service element binds the operations of the corresponding binding to an endpoint address at which clients can invoke this service interface. Based on the relationships between these constructs the static analysis procedure can determine dependencies between BPEL processes onto other BPEL processes and Web services in the following manner.

Listing A.2: Snippets of a corresponding WSDL definition.

```
1 <definitions targetNamespace="http://cs.ucl.ac.uk/omii/bpel/plotutil" ...
2   <types>
3         ...
4   </types>
5
6   <message name="storeResultsRequestMessage">
7     <part name="body" element="tns:storeResultsRequest"/>
8   </message>
9     ...
10
11  <portType name="plotutilPort">
12    ...
13    <operation name="preparePlot">
14      <input message="tns:preparePlotRequestMessage"/>
15    </operation>
16  </portType>
17
18  <binding name="plotutilBinding" type="tns:plotutilPort">
19    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http
          "/>
20    ...
21    <operation name="preparePlot">
22      <soap:operation style="document"/>
23      <input>
24        <soap:body xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" use="literal
              "/>
25      </input>
26      <output>
27        <soap:body xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" use="literal
              "/>
```

```
28      </output>
29    </operation>
30  </binding>
31
32  <service name="plotutil">
33    <port name="plotutilPort" binding="tns:plotutilBinding">
34      <soap:address location="http://huey.cs.ucl.ac.uk:18080/axis/services/
            plotutilPort" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"/>
35    </port>
36  </service>
37
38  <plnk:partnerLinkType name="plotutilPortLink">
39    <plnk:role name="plotutilPortProvider">
40      <plnk:portType name="tns:plotutilPort"/>
41    </plnk:role>
42  </plnk:partnerLinkType>
43 </definitions>
```

1. A given BPEL process, B_1, is parsed and a mapping of prefixes to their corresponding namespaces is created based on the information contained in the preamble of B_1.

2. A second mapping is created. This time from the namespaces to the location of the documents that define these namespaces. This mapping is based on the information contained in the import elements of B_1.

3. Next, the parterLink definitions in B_1 are analysed. For each partnerLink, its name, PLT reference and PLT namespace prefix is recorded. The PLT namespace prefix is resolved to its actual namespace and from that to the location of its corresponding WSDL document. A third mapping is generated. This time from the names of the partnerLinks to the locations of their WSDL documents.

4. For each partnerLink, the corresponding location is accessed to parse its WSDL document, WS_2. WS_2 either defines the PLT of this partnerLink directly or imports another document that defines it.

5. Once the corresponding PLT definition has been found, it is analysed to determine the portType it references. This information is used to look for a binding element with a matching type in WS_2 or in one of its imported documents. The name of the binding is in turn used to look for a matching service element. If such a service element is found, a dependency onto the service at the endpoint URL specified in the service element is recorded.

6. The domain-central part of the process performs an additional step during the merging of the local sub-graphs. If the target namespace of WS_2 matches the target namespace of another BPEL process that has been discovered, then this represents a dependency between two BPEL processes,

namely of B_1 onto BPEL process B_2 whose Web service interface is represented by WS_2. Otherwise, the dependency is between the BPEL process B_1 and the Web service WS_2.

This process is repeated for each partnerLink and for each BPEL process.

The process to determine dependencies from a Web service onto other Web services is based on an analysis of its backend implementation. The deployment descriptors of Web services used by the Axis SOAP runtime [The Apache Software Foundation, 2012a] specify a class file that represents the implementation of the deployed service. The analysis process examines the class files in that location to find references of instances of a particular Axis base class that is used to represent a service instance. These references are usually instantiated on the endpoint URL of the service instance that the backend implementation needs to invoke. The presence of such references indicate that a calling relationship exists between the Web service under analysis and the one represented by the reference. Alternatively, in some cases it is possible to read the endpoint URLs from configuration files that are used by the backend implementation.

Our implementation of the service-level static analysis is based on a number of assumptions. First, it requires that some form of service composition language is used and that this language provides constructs that identify partner services. In the absence of such language constructs, our approach does not work. Second, it assumes that services are statically bound at deployment-time. In case dynamic binding is used, it would be necessary to trace the exchange of messages at runtime to determine dependencies. Third, there needs to be some mechanism that provides information about the location of the interfaces of partner services. For BPEL workflows this can be in the form of import elements. The BPEL modelling environment we have developed [bpe, 2012] makes use of such imports in order to automate the deployment of BPEL workflows. In the absence of such import elements, a BPEL engine may maintain some sort of catalogue to keep track of the interfaces its BPEL processes use. Finally, in order to determine possible calling dependencies between Web services, our method requires access to their backend implementations, unless the endpoint addresses of invoked services can be derived from configuration files. However, access to source code is only required for local resources. The processing of dependencies of remote Web services is left to the Monere agent on that host.

The case study we have presented in Section 5.2 uses BPEL 1.1 to compose its various Web services. As this version of BPEL did not yet allow for the use of import elements, we have hard-coded the service-level dependencies that would otherwise result from the static analysis. However, a proof-of-concept of our static analysis method exists in the form of the so-called ActiveBPEL runtime plugin (ABR), which I have developed as a reference implementation to demonstrate that deployment of BPEL workflows modelled with the Eclipse BPEL Designer can be automated. Version 0.4 of the ABR plugin is available from `http://sse.cs.ucl.ac.uk/projects/past_projects/omii_bpel/downloads/` and the corresponding source code is maintained in the CVS repository of the Software Systems Engineering Group at UCL. The ABR plugin implements a large part of the functionality described above. It can parse a given BPEL process and analyse its imports recursively to find matching WSDL service elements that define the endpoint URLs of the partner services specified in the given

BPEL process. The ABR plugin does not add its results to a dependency model, but instead generates a deployment descriptor, and it does not implement the static analysis of Web service backend implementations to detect calling dependencies between Web services.

Appendix B

Participant Instruction Materials

This appendix contains the instruction materials that participants of the controlled experiment (cf. Chapter 5) were provided with.

B.1 Polymorph Overview

Polymorph Overview

Wednesday, September 29, 2010
7:24 PM

Polymorph Workflow

The Polymorph workflow is a large Web service composition expressing a workflow for experimentation in Theoretical Chemistry. Its input are large datasets describing molecule structures and as its output it calculates the likelihood of a large number of potential molecule arrangements.
It is implemented as a composition of a number of Web services arranged in a number of interacting BPEL processes. BPEL is a workflow language that allows developer to compose basic Web services into larger processes. It contains basic activities to store and manipulate data in variables and to invoke Web service operations.
In general, the BPEL processes in the Polymorph workflow manipulate some input data, use part of it to invoke some functionality via a Web service, then use some form of the resulting data as the input to another Web service operation. The Polymorph workflow also submits compute jobs onto a cluster of Grid compute nodes.

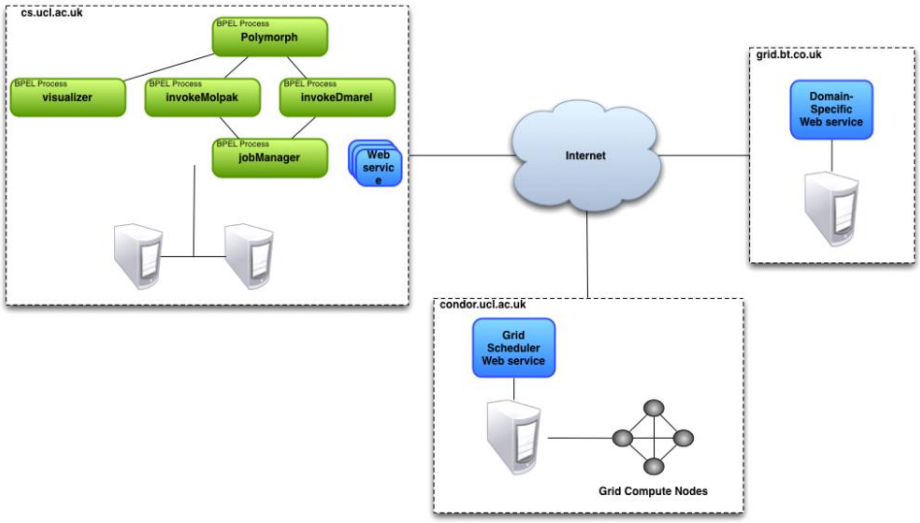
Deployment

The various components of the overall application represented by the Polymorph workflow is deployed across a number of hosts across two different administrative domains.

- Domains:
  - cs.ucl.ac.uk
  - grid.bt.co.uk
- Hosts:
  - dewey.cs.ucl.ac.uk
  - huey.cs.ucl.ac.uk
  - louie.cs.ucl.ac.uk
  - cpu029.grid.bt.co.uk (217.33.61.113)

BPEL Processes

A brief overview of the various BPEL processes comprising the Polymorph workflow starting with the leaf-level processes. In simple terms, the Polymorph workflow is about executing many parallel instances of two domain-specific executables on Grid compute nodes (first MOLPAK and then DMAREL on the output of MOLPAK) and then formatting the resulting data in a domain-specific manner for display on an HTML page.



visualizer -- This BPEL process is called on the data resulting from a run of the Polymorph workflow and produces the final output of the workflow as a number of files.

- Receives resulting data and submits it to the Graph service, which is hosted by the remote administrative domain, in order to format the results in a specific manner.
- Then stores formatted results as a png image and archive and generates a html file displaying formatted results. The directory in which these files are stored is /home/omii_tomcat/omii-uk-server/webapps/axis/polymorph on huey.cs.ucl.ac.uk.

jobManager -- This BPEL process accepts descriptions of compute jobs in XML, submits them to the Grid and monitors their execution.

- Upon receipt of a job description in XML, the jobManager process invokes the GridSAM Web service (discussed in later section) to submit the compute job to a Grid scheduler.
- The jobManager process continues to poll the GridSAM Web service for the status of its submitted jobs until their status is returned as completed or failed.
- The jobManager process then returns the resulting data from the compute job to the calling BPEL process.

molpak -- Extracts required elements from input data and prepares submission of molpak compute jobs on the Grid.

- Manipulates input data it has received from molpakdmarel and prepares an XML job description of a corresponding compute jobs.
- Invokes jobManager process instances to actually submit job onto compute Grid and waits for response.
- Upon receiving successful response, returns data to molpakdmarel process.

dmarel -- Operates in exactly the same manner as the molpak process, but submits Grid compute jobs for the DMAREL executable.

molpakdmarel -- This is the top-level BPEL process that coordinates among all the other sub-processes.

- Upon receipt of input data, it invokes one ore more instances (depending on the input) of the molpak BPEL process.
- As each molpak process instance returns its resulting data, molpakdmarel invokes 100 instances of the dmarel process, which also run concurrently.
- Once the dmarel instances have all completed and returned their resulting data, molpakdmarel invokes the visualizer process and completes.

In addition to the above steps, these BPEL processes make use of a number of utility Web services, described below.

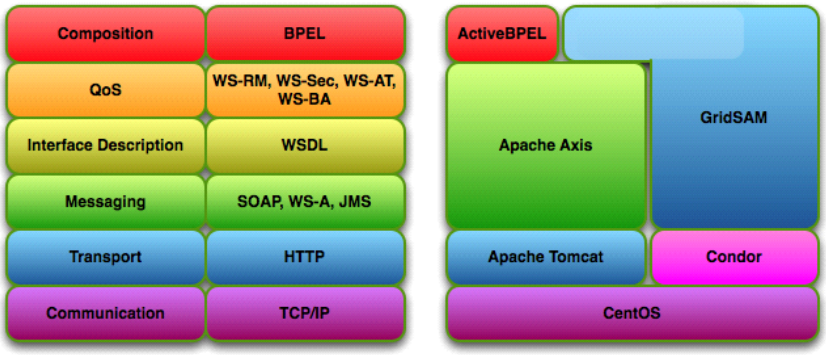
Web Services

There are a number of SOAP Web services used by the Polymorph workflow. The first group of Web services are the actual Web service interfaces of the BPEL processes. Each BPEL process has a Web service interface through which it interacts with other BPEL processes and Web services. These are all deployed on the ActiveBPEL engine on dewey.cs.ucl.ac.uk and are named after the BPEL processes they represent.

dewey.cs.ucl.ac.uk	▪ molpakdmarelPartnerService ▪ molpakPartnerService ▪ dmarelPartnerService ▪ jobmanagerPartnerService ▪ visualizer
huey.cs.ucl.ac.uk	• plotutilPort -- has operations to submit results to remote Graph service (obtain scatter plot of results) and store results (as HTML file). • polyutilPort -- creates an archive of all resulting data and stores it on huey. • Molpak2CMLPort -- has operations to convert input data and interim results into various formats. • gridsam -- has operations to submit job descriptions, get jobs executed on Grid compute nodes and monitor job status.
217.33.61.113	• Graph -- has operations to create scatter plot of submitted data points. This service is accessible from the CS domain under the following URL: http://217.33.61.113:18080/PlotWS/services/Graph.

Middleware

The following figure gives an overview of the middleware components supporting the Polymorph service compositions and their main function.



The infrastructure and implementation of the Graph service in the grid.bt.co.uk (217.33.61.113) domain is not known to us as it resides in a different administrative domain.

dewey.cs.ucl.ac.uk	Operating system	• CentOS 5
	Servlet container	• Apache Tomat • http://dewey.cs.ucl.ac.uk:18080/ • Tomcat JVM 1.5
	SOAP runtime	• Apache Axis deployed in Tomcat instance

	BPEL engine	• ActiveBPEL deployed in Tomcat instance
huey.cs.ucl.ac.uk	Operating system	• CentOS 5
	Servlet container	• Apache Tomcat • http://huey.cs.ucl.ac.uk:18080/
	SOAP runtime	• Apache Axis deployed in Tomcat instance
	Grid middleware	• Gridsam deployed in Tomcat instance
	Grid middleware	• Condor running as a set of processes as root
cs.ucl.ac.uk	Grid compute nodes	• Around 60 64-bit Linux machines in student labs available for executing Grid compute jobs

GridSAM consists of a Web service interface that allows BPEL process/Web services to submit and monitor descriptions of Grid compute jobs to the GridSAM server, which translates said job descriptions into a format suitable for interaction with Condor.

Condor is a Grid job scheduler that handles queuing and scheduling of compute jobs onto Grid compute nodes and makes their results, if any, available to its clients.

End-To-End Operation

Users set up and start new runs of the Polymorph workflow via a web application (http://huey.cs.ucl.ac.uk:18080/PolyClient/Invoke.htm). This web application constructs a suitable input message to the top-level BPEL process, molpakdmarel, and then display a page that allows end users to follow the progress of their just started run. When a run completes (after several minutes, hours or days depending on the input dataset), users obtain the result files on huey.cs.ucl.ac.uk in /home/omii_tomcat/omii-uk-server/webapps/axis/polymorph/. The result files are a png file containing a scatter plot of the results, a HTML page displaying results in tabular format and including the png and an archive of these files.

B.2 Control Group Instructions

Control Group User Guide

Introduction

The goal of this experiment is to quantify the time to diagnose certain types of failures. We will run the Polymorph application 10 times, each time injecting a randomly selected failure into the system. I will then notify you of a complaint by the user about incorrect service/functioning of the latest run of the Polymorph workflow. Given this information, you will act as the administrator responsible for identifying the cause of the problem using the tools available to you.

Shell

You will have root-level access to huey.cs.ucl.ac.uk and dewey.cs.ucl.ac.uk via a number of shell sessions. You are encouraged to run any commands you need in order to determine the cause of a failure.

Log Files

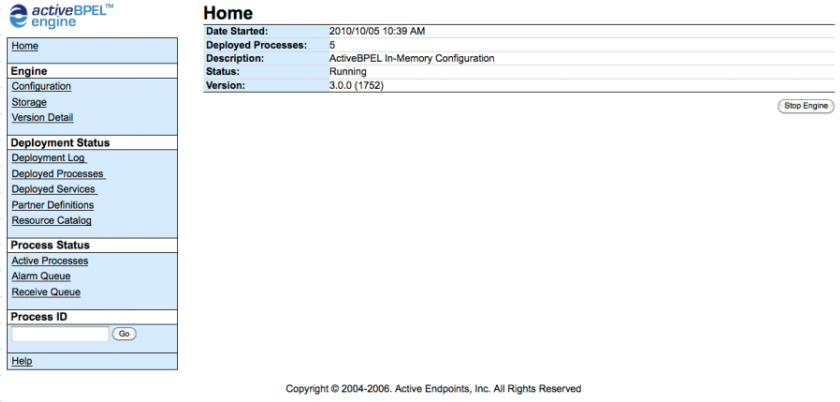
Following are the locations of a number of log files:

OS logs on huey and dewey	/var/log
Tomcat log on huey and dewey	/home/omii_tomcat/omii-uk-server/jakarta-tomcat-5.0.28/logs/catalina.out
Gridsam log on huey	/home/omii_tomcat/omii-uk/server/jakarta-tomcat-5.0.28/logs/gridsam.log
Condor logs on huey	/home/condor/log

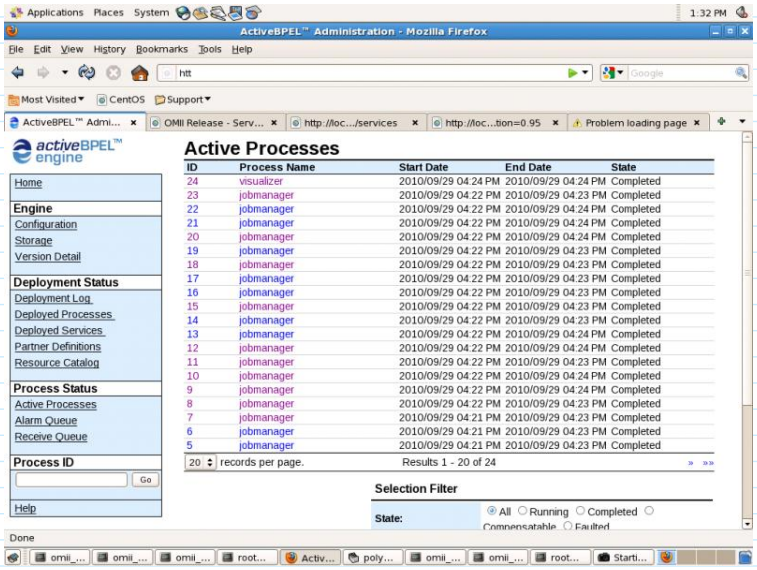
ActiveBPEL Monitoring Console

The ActiveBPEL engine provides a web-based monitoring console, which allows you to monitor the progress of the various BPEL processes during a run. It can be accessed in your browser at http://dewey.cs.ucl.ac.uk:18080/BpelAdmin.

The home screen of the monitoring console:

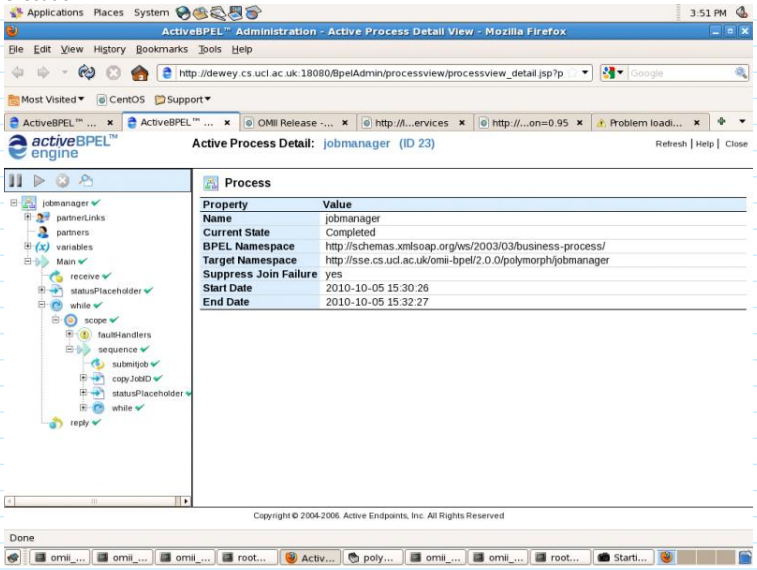


Click on Active Processes to see a list of currently running process instances:

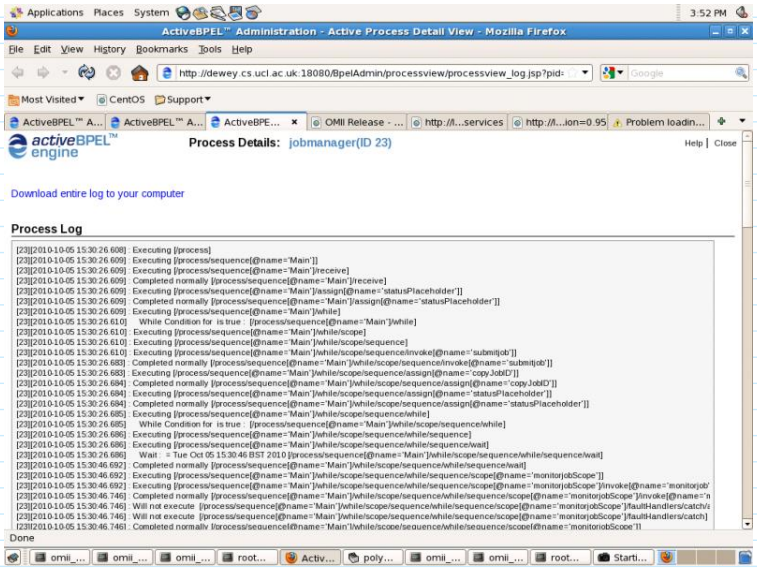


The list shows the process instance id, name, start and finish time as well as its status (running, faulted, completed).

Click on the any of the process instances to retrieve a graphical representation of its current state in the execution:



The green arrows indicate activities this particular BPEL process has completed successfully. A red 'X' indicates unsuccessful activities.
Click on the right-most icon atop the graphical representation to inspect the process log file:



The log file captures a step-by-step description of all events the selected BPEL process is involved in.

Condor

There are two important commands to inspect the performance of the Condor Grid job manager.

- condor_q -- shows how many compute jobs are in the queue, how many are running and idle
- condor_status -- shows information about available compute nodes

JConsole

You will have access to a JConsole session on huey and dewey monitoring the Tomcat JVMs on these hosts.

Diagnosis Process

The desired diagnosis should answer what and where. That is, it should identify the actual component(s) responsible for the failure and what about their behaviour is abnormal (i.e. which metric is out of bounds). Such a result is a high-level determination of a cause, such as for example that a particular component exhausted some resource or is very busy. The detailed diagnosis would be carried out with more specialised tools (i.e. Java profiler) outside of this experiment.

B.3 Effect Group Instructions

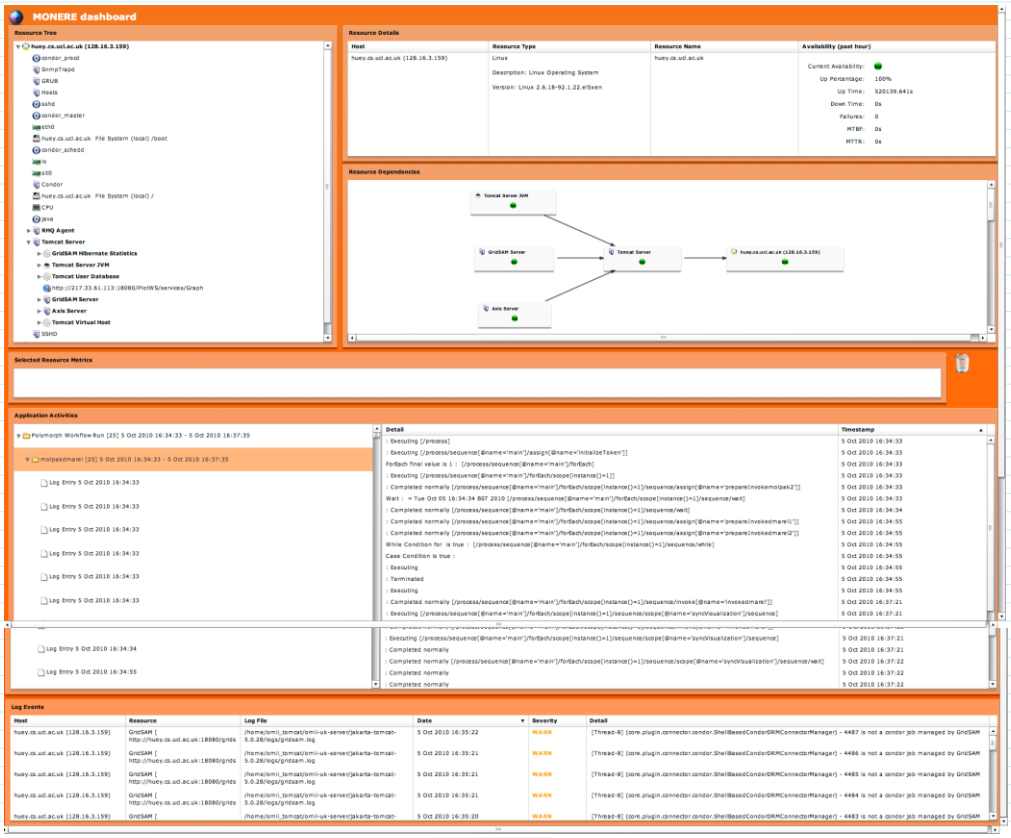
Monere User Guide

Introduction

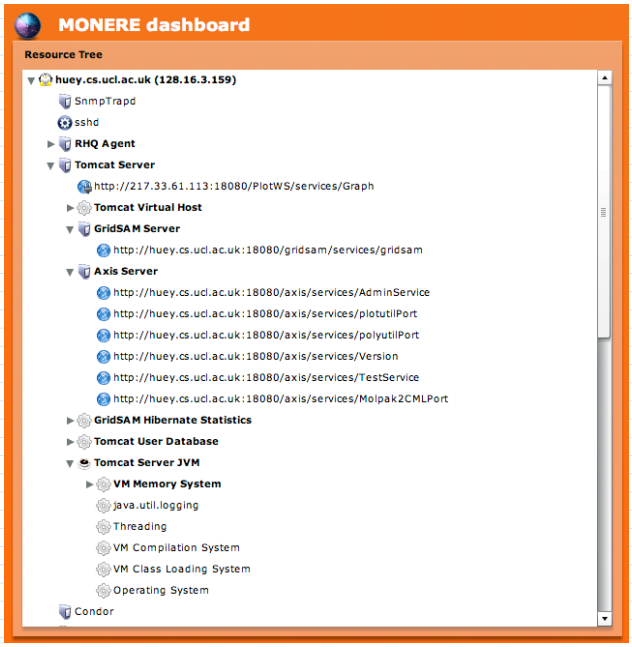
The goal of this experiment is to quantify the time to diagnose certain types of failures. We will run the Polymorph application several times, each time injecting a randomly selected failure into the system. I will then notify you of a complaint by the user about incorrect service/functioning of the latest run of the Polymorph workflow. Given this information, you will act as the administrator responsible for identifying the cause of the problem using the Monere monitoring framework.

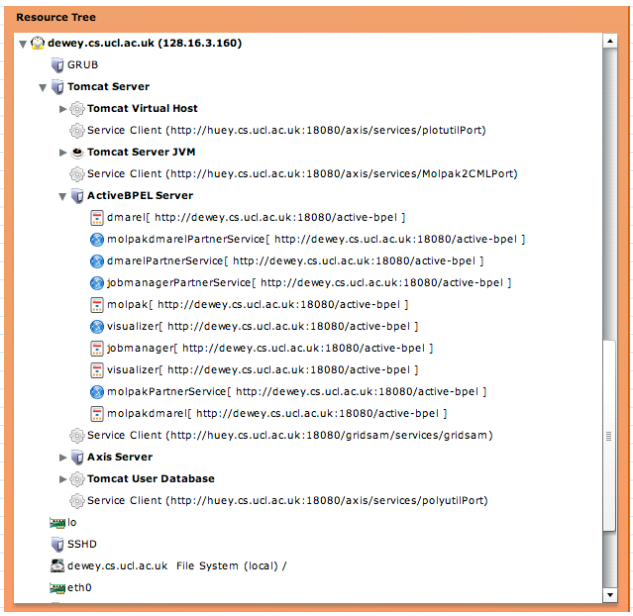
UI Overview

Monere is a monitoring framework that discovers all dependencies of a given application and then deploys a set of agents to continuously collect certain metrics about the discovered components. The picture below shows an overview of Monere's user interface, which provides access to the collected information. The following sections discuss the various panels in the UI.



Resource Tree

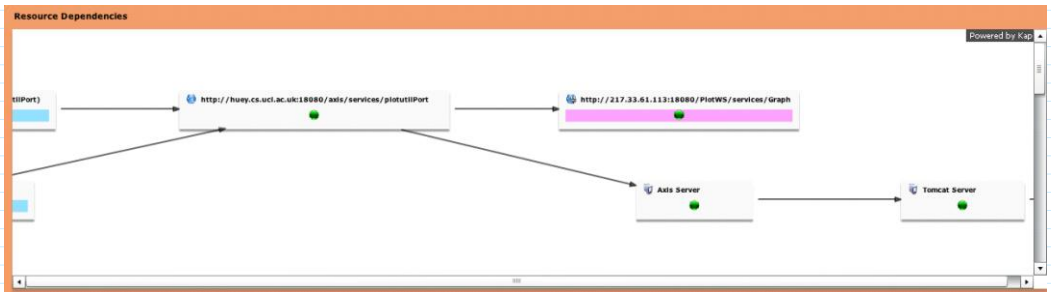




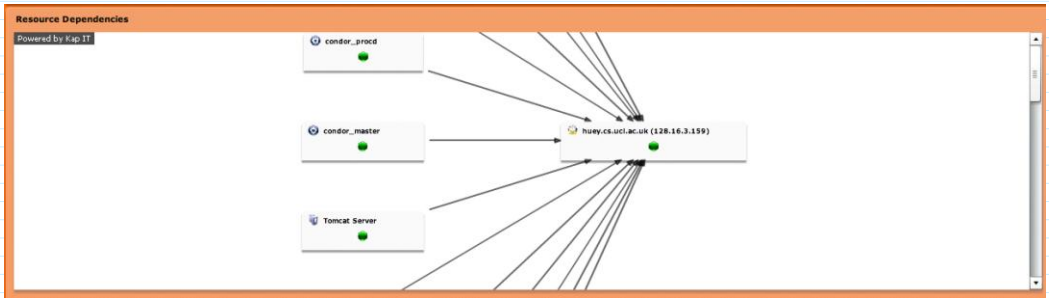
The Resource Tree lists all resources that have been discovered and for which measurements are collected. The display is hierarchical with a host being the root and can be expanded by clicking on the arrow next to some resources.
The different icons represent different types of resources. Each type of resource provides a set of metrics about that resource.

Resource Details
Selecting a resource in the Resource Tree by clicking on it, will display more information about it in the Resource Details view. The information includes the host of this resource, the type of resource and its name and some information about its current availability. A green dot indicates that the selected resource is currently available; a red dot means that it is currently unavailable.

Resource Dependencies
Selecting a resource in the Resource Tree by clicking on it, will also visualize it along with its dependencies in the Resource Dependencies view. This shows resources that directly depend on the selected resource and all resources that it depends on. Resources on different hosts are displayed with different colour highlighting. Green and red dots represent current availability of a resource.

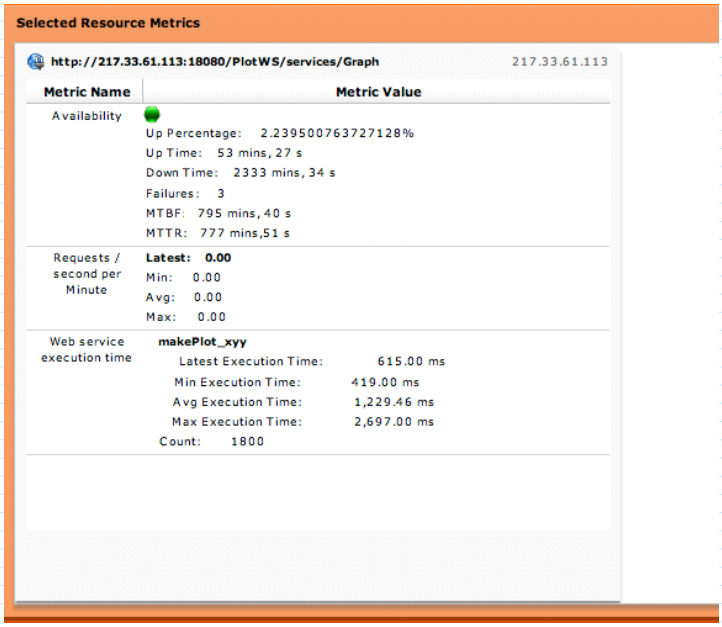


Clicking on another resource in the Resource Dependencies view will retrieve and display the dependencies of this resource. For example, if we click on dewey.cs.ucl.ac.uk in the view above, the result is a view of its dependencies:



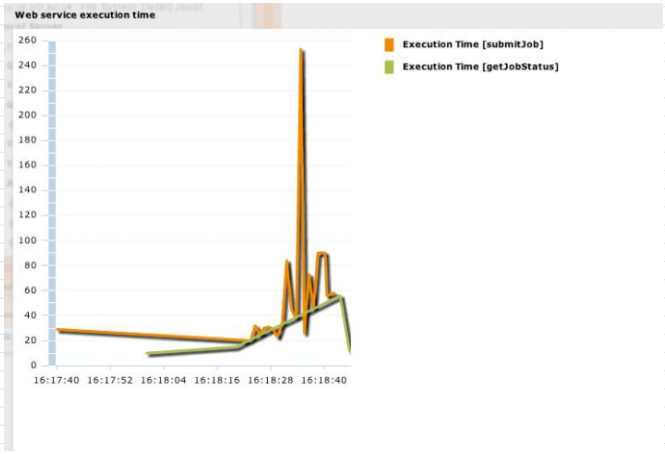
If the dependency graph is too large to view, you can use the mouse pointer to drag your view of it around.

Selected Resource Metrics
You can drag resources from the Resource Tree onto the field below called Selected Resource Metrics. This will display a table with its current metrics. There is one row per metric. Several components can be dragged onto this view and will be displayed next to each other.
Each resource has a different set of metrics. In this example we see information about the availability of this resource, its requests per second (there's a bug, this will often display 0) and the execution time of this Web service along with the number of requests it has served (i.e. 1,800). In general, displayed values are aggregated over the past 60 minutes along with information about the latest measurement.
Please note that if a component has become unavailable, the metrics and charts displayed will be for the last measurement collection before unavailability. That is, in case of unavailability you may see the state of the component as it was up to 30 seconds prior to unavailability in the worst case. In such cases in can be helpful to chart a metric in order to get an impression of any recent trends.

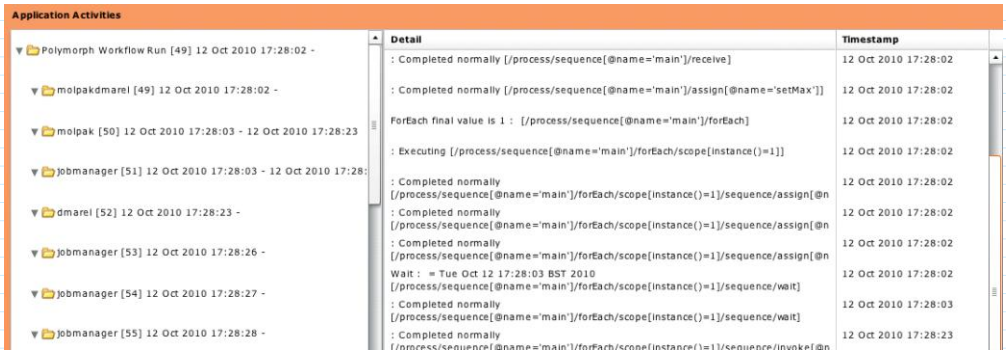


Sometimes, a row may contain more information than can be fully displayed. In that case, please use the arrow keys to scroll up and down within a row, after having selected the row.
To remove a resource from the Selected Resource Metrics view very carefully drag it over to the garbage can to the right of this view.

Chart View
For many of the metrics historical data can be accessed and displayed as a chart by double-clicking on the row containing a particular metric. The chart opens in a separate pop-up window.



Application Activities
The Application Activities view displays information about the current run of the Polymorph workflow. On the left-hand side are the BPEL process-instances invoked by the current run along with their start and end times. Clicking on any of the process instances will display a detailed view of the activities performed to date by the selected instance on the right-hand side. Any activities (on the right-hand side) that have experienced some kind of fault will be highlighted in red.



Time Window
Double-clicking on any process instance or activity detail in the Application Activities view will display a so-called Time Window. A Time Window is a replica of the Monere UI that only displays data for a short period of time around the start and end time of the selected process instance or activity detail. This feature can be useful, if you need to determine what measurements were recorded by some resources at around the time of a specific application activity. The Time Window has the same views as described here and works in exactly the same way.

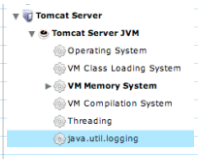
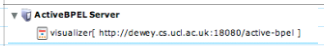
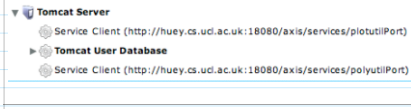
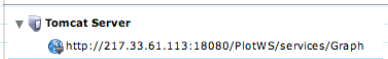
Log Events
Finally, the log events view displays logging events of severity WARN and ERROR from multiple sources, such as the Tomcat instances, the operating systems logs, GridSAM and Condor.

It displays the host on which the event originated, the corresponding resource and log files, a timestamp, the severity and some information about the event.
To sort the information by timestamp, it is necessary to manually click on the Date field twice.

Resource Types and Metrics

Following is an overview of the various resource types and the metrics they expose. This is not a complete listing of all the metrics that can be inspected for each resource type. All resource types maintain data about their past and current availability.

Resource Type	Description	Metrics
huey.cs.ucl.ac.uk (128.16.3.159)	Host/operating system	• Free Memory • Used Memory • Free Swap Space • Total Swap Space • Swap Space In --> and indication of the swap activity on this host • Swap Space Out • Idle, System, User, Wait Load --> CPU load • Maximum file descriptors system-wide and per-process --> OS limits on file descriptors • Maximum file descriptors per process • Maximum number of processes system-wide --> OS limits on number of running processes • Currently open file descriptors --> a listing of the number of open file descriptors per OS user • Currently running processes --> a listing of the number of currently running OS processes per OS user
CPU	CPU	• The usual suspects
huey.cs.ucl.ac.uk File System (local) /boot	File system	• Disk reads and writes per minute --> an indication of actual I/O activity • Free Space • Used Space • Used Percentage
eth0	Network Adapter	• Bytes/Packetes received/transmitted per minute • Receive/transmit packets dropped per minute • Latency to destination --> the average and latest latencies experienced between this network interface and the network interface of a host we depend on
Tomcat Server	Server (Tomcat, Axis, ActiveBPEL, etc.)	• These resource types usually don't have their own metrics, but instead contain child resources that expose relevant metrics.

	Tomcat Server JVM	• The various child resources contain metrics such as: • Number of started, suspended, deadlocked threads (Threading) • Heap space usage (VM Memory System) • Loaded and unloaded classes (Class Loading) • And so on.
	Web service	• Requests/second --> not working reliably • Web service execution time --> average execution times per Web service operation and execution counts
	BPEL process	• BPEL process execution time
	Service client --> a dependency from a component on one host (in this case dewey) onto a named Web service on another host	• Availability --> from the viewpoint of clients on this host • Web service client response time --> response times experienced by clients on this host of the named Web service
	Interdomain component -- a dependency from a component on this host (in this case huey) onto a named Web service in a remote administrative domain	• Availability --> from the viewpoint of clients on this host • Web service execution time --> average and latest execution times of the remote Web service sorted by Web service operation
	OS process	• CPU usage • Kernel Time per Minute • User Time per Minute • Physical Memory • Virtual Memory • Current threads • Current open file descriptors
	Grid job manager	• Condor Jobs --> number of submitted, completed, failed jobs and their average duration • Condor Nodes --> overview of the availability of compute nodes • Condor Queue --> number of jobs in the queue sorted by their state (i.e. running, idle, held)

Diagnosis Process

The Monere framework suggests a certain process when diagnosing the cause of a failure. The UI will not necessarily be able to point you directly to the cause of a failure. However, its different views offer some guidance to determining the cause of a failure.

- Inspect the Log Events to find any obvious error messages that should be further investigated.
- Inspect the Application Activities to see whether any process instances have reported failures that should be further investigated.
- Select relevant resources (for example, the jobManager BPEL process) to obtain an overview of its dependencies.
- Check the availability of these dependencies.
- Then drill further down by inspecting the metrics of resources along relevant paths of dependencies to find any anomalies in the metrics that might explain the cause of the problem.

The desired diagnosis should answer what and where. That is, it should identify the actual component(s) responsible for the failure and what about their behaviour is abnormal (i.e. which metric is out of bounds). Such a result is a high-level determination of a cause, such as for example that a particular component exhausted some resource or is very busy. The detailed diagnosis would be carried out with more specialised tools (i.e. Java profiler) outside of this experiment.

Following is a walkthrough example of a failure diagnosis using the Monere framework.

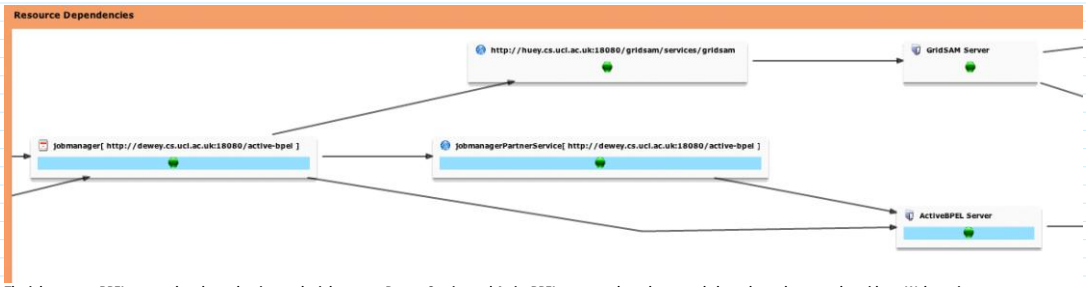
Example 1:

In this example the end user complains that the latest run seems to be making no progress and is stuck running jobManager process instances.
First, we check the Log Events, but find no error messages that could explain this failure.
Next, we inspect the Application Activities. We find that the jobManager instances keep looping and repeating an invocation to some Web service.

Activites View shows jobmanager BPEL process instances looping.

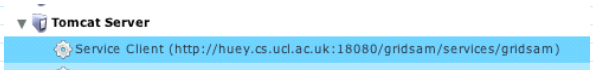
We select the jobManager BPEL process resource in the Resource Tree to view its dependencies. They all appear to be available. We can see that the jobManager

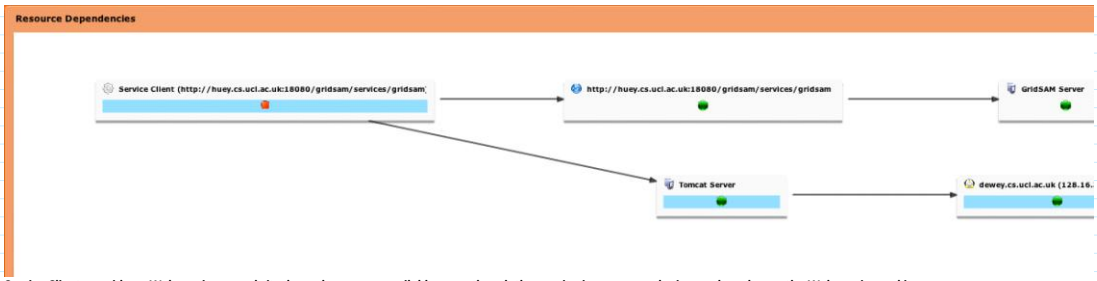
BPEL process on host dewey has a dependency on the GridSAM Web service on host huey.



The jobmanager BPEL process has dependencies on the jobmanagerPartnerService and ActiveBPEL server on host dewey and also a dependency on the gridsam Web service on host huey.

Next, we choose to select the Service Clients on dewey to view their dependencies. We find that all their dependencies, including the one on the GridSAM Web service appear to be unavailable. That is, even though these resources appear available from the perspective of the monitoring server, they are not reachable by clients on host dewey.



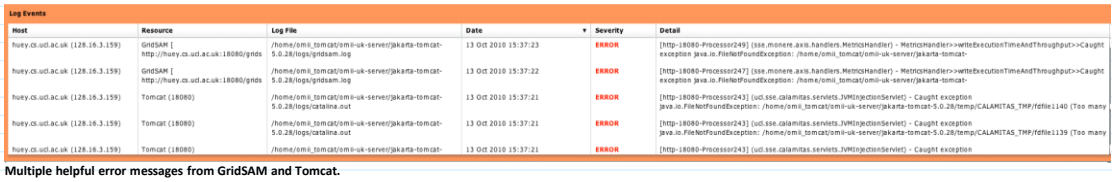


Service Client on gridsam Web service regards its dependency as unavailable, even though the monitoring server on louie.cs.ucl.ac.uk sees the Web service and its dependencies as available.

We suspect a network issue and drag the network interface resource on host dewey onto the Selected Resource Metrics View. We find that the network interface on dewey is experiencing very high latencies to host huey and conclude that the cause of the reported failure is this heightened latency due to some network problem.

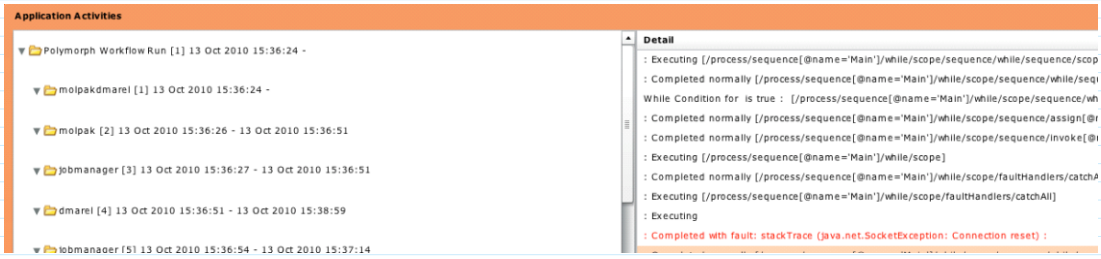
Example 2:

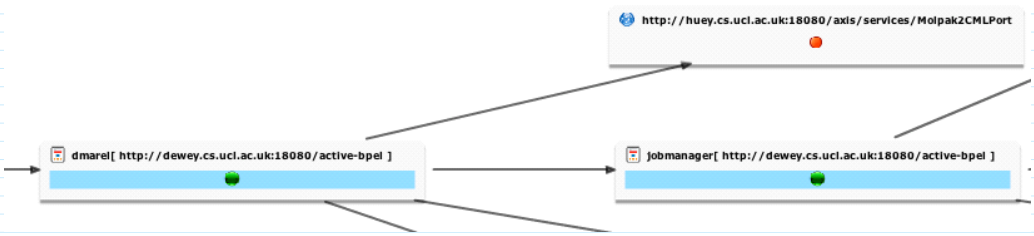
The end user complains that the workflow is not making any progress anymore.
First, we look at the Log Events and find that GridSAM on huey couldn't write a file and may find that Tomcat on huey has experienced a SocketException: too many open files. We know what the problem is, but need to investigate further to exactly determine the responsible component and offending metric.

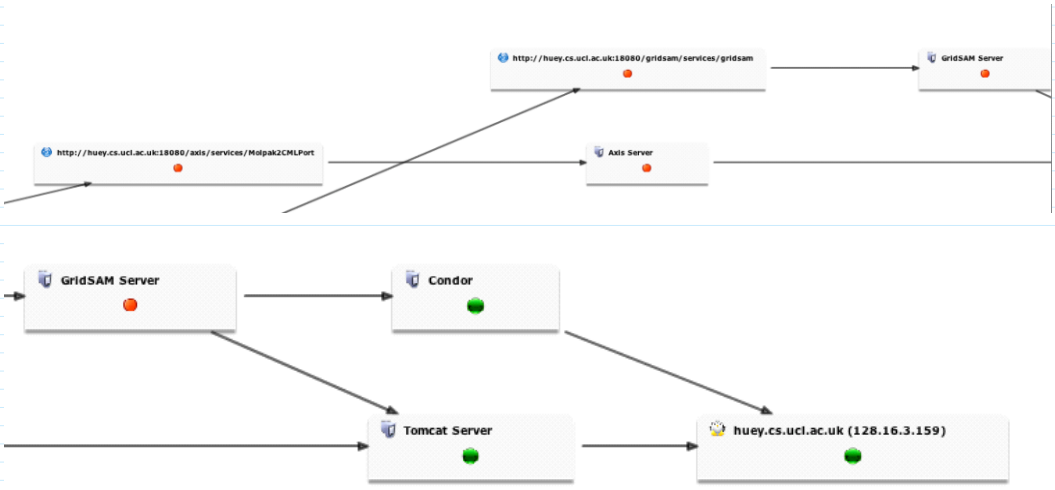


Multiple helpful error messages from GridSAM and Tomcat.

The Application Activities view reveals failed process instances and we select one of these, say the dmarel BPEL process on dewey, from the Resource Tree to view its dependencies. We find that all of its dependencies on huey apart from the Tomcat server and OS seem to be unavailable.

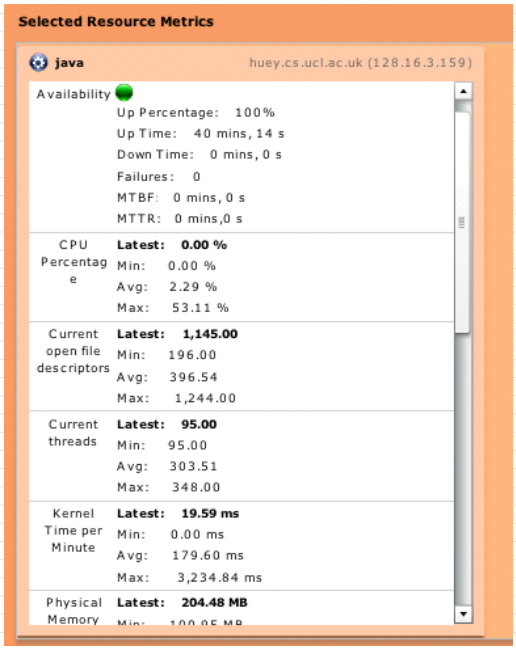






We begin to drag the dependencies on huey onto the Selected Resource Metrics view. We find that the GridSAM Web service is unavailable, but its metrics appear normal. Next, we drag the the children of the Tomcat Server JVM resource into the Selected Resource Metrics view in order to inspect metrics, such as heap usage, threading and class loading. All these metrics appear to be within normal bounds.

Finally, we inspect the metrics of the Java OS process on huey and indeed find a recent large increase in the number of open file descriptors



Bibliography

[AWS, 2008] (2008). AWS outage. https://forums.aws.amazon.com/thread.jspa?threadID=20932&start=0&tstart=0. accessed Aug 04, 2012.

[goo, 2009] (2009). GMail outage. http://googleblog.blogspot.co.uk/2009/02/update-on-gmail.html. accessed Aug 04, 2012.

[Fac, 2010] (2010). Facebook outage. http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919. accessed Aug 04, 2012.

[iso, 2011] (2011). ISO/IEC/IEEE 42010-2011 FAQ. http://www.iso-architecture.org/ieee-1471/faq.html.

[bpe, 2012] (2012). Eclipse BPEL Designer. http://www.eclipse.org/bpel/. accessed May 20, 2012.

[fou, 2012] (2012). foursquare. https://foursquare.com/. accessed Aug 06, 2012.

[hp:, 2012] (2012). HP Network Management Center. http://www8.hp.com/us/en/software-solutions/software.html?compURI=1171412. accessed Aug 23, 2012.

[ibm, 2012] (2012). IBM Tivoli. http://www-01.ibm.com/software/tivoli/. accessed Aug 23, 2012.

[nag, 2012] (2012). Nagios. http://www.nagios.org/. accessed Aug 23, 2012.

[red, 2012] (2012). reddit. http://www.reddit.com/. accessed Aug 06, 2012.

[tcp, 2012] (2012). tcpdump. http://www.tcpdump.org/. accessed Apr 16, 2012.

[Active Endpoints, 2012a] Active Endpoints (2012a). ActiveBPEL. http://www.activebpel.org. accessed Apr 03, 2012.

[Active Endpoints, 2012b] Active Endpoints (2012b). ActiveBPEL Admin API. http://www.activevos.com/learn/open-source. accesssed Mar 15, 2012.

[Agarwala et al., 2007] Agarwala, S., Alegre, F., Schwan, K., and Mehalingham, J. (2007). E2EProf: Automated End-to-End Performance Management for Enterprise Systems. In *Proceedings of the 37th*

Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, pages 749–758, Washington, DC, USA. IEEE Computer Society.

[Agarwala and Schwan, 2006] Agarwala, S. and Schwan, K. (2006). SysProf: Online Distributed Behavior Diagnosis through Fine-grain System Monitoring. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems.*, ICDCS 2006, page 8.

[Aguilera. et al., 2000] Aguilera., M. K., Chen, W., and Toueg, S. (2000). On Quiescent Reliable Communication. *SIAM Journal on Computing*, 29(6):2040–2073.

[Al-Shaer et al., 1999] Al-Shaer, E., Abdel-Wahab, H., and Maly, K. (1999). HiFi: a new monitoring architecture for distributed systems management. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems.*, ICDCS 1999, pages 171 –178.

[Andrews et al., 2003] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. (2003). Business Process Execution Language for Web services (BPEL4WS) Version 1.1. Technical report, OASIS. http://public.dhe.ibm.com/software/dw/specs/ws-bpel/ws-bpel.pdf.

[Anjomshoaa et al., 2005] Anjomshoaa, A., Drescher, M., Fellows, D., Pulsipher, D., and Savva, A. (2005). Job Submission Description Language (JSDL) Specification version 1.0. Technical report, Open Grid Forum. http://www.gridforum.org/documents/GFD.56.pdf.

[Ardissono et al., 2006] Ardissono, L., Furnari, R., Goy, A., Petrone, G., and Segnan, M. (2006). Fault tolerant web service orchestration by means of diagnosis. In *Proceedings of the Third European Conference on Software Architecture*, EWSA 2006, pages 2–16, Berlin, Heidelberg. Springer-Verlag.

[Arkin et al., 2002] Arkin, A., Askary, S., Fordin, S., Jekeli, W., Kawaguchi, K., Orchard, D., Pogliani, S., Riemer, K., Struble, S., Nagy, P. T., and amd Sinisa Zimek, I. T. (2002). Web Services Choreography Interface (WSCI) 1.0. W3C note, W3C. http://www.w3.org/TR/wsci/.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11 – 33.

[Baresi and Guinea, 2011] Baresi, L. and Guinea, S. (2011). Self-Supervising BPEL Processes. *IEEE Transactions on Software Engineering*, 37(2):247 –263.

[Barghouti and Kaiser, 1991] Barghouti, N. S. and Kaiser, G. E. (1991). Concurrency control in advanced database applications. *ACM Computing Surveys*, 23:269–317.

[Barham et al., 2004] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. (2004). Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design & Implementation*, OSDI 2004, pages 18–18, Berkeley, CA, USA. USENIX Association.

[Bartolini et al., 2009] Bartolini, C., Bertolino, A., Marchetti, E., and Polini, A. (2009). WS-TAXI: A WSDL-based Testing Tool for Web Services. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation.*, ICST 2009, pages 326 – 335.

[Basu et al., 2008] Basu, S., Casati, F., and Daniel, F. (2008). Toward Web Service Dependency Discovery for SOA Management. In *Proceedings of the 2008 IEEE International Conference on Services Computing.*, volume 2 of *SCC 2008*, pages 422 –429.

[Bausch, 2004] Bausch, W. (2004). *OPERA-G : a microkernel for computational grids*. Dissertation, Doctor of Technial Sciences, Swiss Federal Institute of Technology Zurich.

[Bertolino et al., 2006] Bertolino, A., Frantzen, L., Polini, A., and Tretmans, J. (2006). Audition of Web Services for Testing Conformance to Open Specified Protocols. In Reussner, R., Stafford, J., and Szyperski, C., editors, *Architecting Systems with Trustworthy Components*, number 3938 in Lecture Notes in Computer Science, pages 1–25. Springer.

[Best and Rayner, 1987] Best, D. J. and Rayner, J. C. W. (1987). Welch's Approximate Solution for the Behrens-Fisher Problem. *Technometrics*, 29(2):pp. 205–210.

[Bhatia et al., 2008] Bhatia, S., Kumar, A., Fiuczynski, M. E., and Peterson, L. (2008). Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, OSDI 2008, pages 103–116, Berkeley, CA, USA. USENIX Association.

[Birman and Renesse, 1994] Birman, K. P. and Renesse, R. V. (1994). *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA.

[Bozkurt et al., 2010] Bozkurt, M., Harman, M., and Hassoun, Y. (2010). Testing Web Services: A Survey. Technical Report TR-10-01, Department of Computer Science, King's College London.

[Bray et al., 2008] Bray, T., Paoli, J., Maler, E., Yergeau, F., and Sperberg-McQueen, C. M. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C recommendation, W3C. http://www.w3.org/TR/2008/REC-xml-20081126/.

[Brown et al., 2001] Brown, A., Kar, G., and Keller, A. (2001). An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proceedings of the 2001 IEEE/IFIP International Symposium on Integrated Network Management*, IM 2001, pages 377 –390.

[Candea, 2004] Candea, G. (2004). Predictable Software - A Shortcut to Dependable Computing? Technical report, Stanford University. http://arxiv.org/abs/cs.OS/0403013.

[Candea et al., 2003] Candea, G., Delgado, M., Chen, M., and Fox, A. (2003). Automatic failure-path inference: a generic introspection technique for Internet applications. In *Proceedings of the 3rd IEEE Workshop on Internet Applications.*, WIAPP 2003, pages 132 – 141.

[Candea et al., 2004] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. (2004). Microreboot - A technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, OSDI 2004, pages 3–3, Berkeley, CA, USA. USENIX Association.

[Cantrill et al., 2004] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. (2004). Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, ATEC 2004, pages 15–28, Berkeley, CA, USA. USENIX Association.

[Carrozza et al., 2008] Carrozza, G., Cotroneo, D., and Russo, S. (2008). Software Faults Diagnosis in Complex OTS Based Safety Critical Systems. In *Proceedings of the 7th European Dependable Computing Conference.*, EDCC 2008, pages 25–34.

[CentOS, 2012] CentOS (2012). CentOS. http://wiki.centos.org/. accessed Apr 03, 2012.

[Chan et al., 2006] Chan, P. P. W., Lyu, M. R., and Malek, M. (2006). Making services fault tolerant. In *Proceedings of the 3rd International Conference on Service Availability*, ISAS 2006, pages 43–61, Berlin, Heidelberg. Springer-Verlag.

[Chandola et al., 2009] Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM Computing Surveys*, 41:15:1–15:58.

[Chandra et al., 2008] Chandra, A., Prinja, R., Jain, S., and Zhang, Z. (2008). Co-designing the failure analysis and monitoring of large-scale systems. *SIGMETRICS Performance Evaluation Review*, 36:10–15.

[Chandra and Toueg, 1996] Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

[Chen and Avizienis, 1995] Chen, L. and Avizienis, A. (1995). N-Version Pogramming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing - Volume III,'Highlights from Twenty-Five Years'.*, FTCS-25, pages 113–119. IEEE.

[Chen et al., 2002] Chen, M., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. (2002). Pinpoint: problem determination in large, dynamic Internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks.*, DSN 2002, pages 595 – 604.

[Chen et al., 2008] Chen, X., Zhang, M., Mao, Z. M., and Bahl, P. (2008). Automating network application dependency discovery: experiences, limitations, and new solutions. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 117–130, Berkeley, CA, USA. USENIX Association.

[Chen, 2008] Chen, Y. (2008). *WS-Mediator for Improving Dependability of Service Composition*. PhD thesis, Newcastle University.

[Christensen et al., 2001] Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001). Web Services Description Language (WSDL) 1.1. W3C note, W3C. http://www.w3.org/TR/wsdl.

[Clark and DeRose, 1999] Clark, J. and DeRose, S. (1999). XML Path Language (XPath) 1.0. W3C recommendation, W3C. http://www.w3.org/TR/xpath/.

[Clement et al., 2005] Clement, L., Hately, A., von Riegen, C., and Rogers, T. (2005). Universal Description, Discovery and Integration (UDDI) 3.0.2. OASIS standard, OASIS. https://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm.

[Cohen et al., 2004] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S. (2004). Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, OSDI 2004, pages 16–16, Berkeley, CA, USA. USENIX Association.

[Cohen, 1960] Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46.

[Crockford, 2006] Crockford, D. (2006). The application/json Media Type for JavaScript Object Notation (JSON). Technical Report 4627, Internet Engineering Task Force. RFC 4627 (Informational).

[Curbera et al., 2002] Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., and Weerawarana, S. (2002). Business Process Execution Language for Web services (BPEL4WS) Version 1.0. Technical report. http://public.dhe.ibm.com/software/dw/specs/ws-bpel/ws-bpel1.pdf.

[Davis et al., 2009] Davis, D., Karmarkar, A., Pilz, G., Winkler, S., and Yalcinalp, U. (2009). Web Services Reliable Messaging (WS-RM) Version 1.2. Technical report, OASIS. http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.pdf.

[De Pauw et al., 2005] De Pauw, W., Lei, M., Pring, E., Villard, L., Arnold, M., and Morar, J. F. (2005). Web services navigator: visualizing the execution of web services. *IBM Systems Journal*, 44(4):821–845.

[Dempster et al., 1977] Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):pp. 1–38.

[Elmagarmid, 1992] Elmagarmid, A. K., editor (1992). *Database Transaction Models for Advanced Applications*. Morgan Kaufmann.

[Elmagarmid et al., 1990] Elmagarmid, A. K., Leu, Y., Litwin, W., and Rusinkiewicz, M. (1990). A Multidatabase Transaction Model for InterBase. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 507–518, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Emmerich, 2000] Emmerich, W. (2000). *Engineering Distributed Objects*. John Wiley & Sons, Chichester, UK.

[Emmerich et al., 2005] Emmerich, W., Butchart, B., Chen, L., Wassermann, B., and Price, S. L. (2005). Grid Service Orchestration using the Business Process Execution Language (BPEL). *Journal of Grid Computing*, 3(3-4):283–304.

[Feingold and Jeyaraman, 2009] Feingold, M. and Jeyaraman, R. (2009). Web Services Coordination (WS-Coordination) Version 1.2. OASIS standard, OASIS. http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html.

[Ferrante et al., 1987] Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349.

[Fielding and Taylor, 2002] Fielding, R. T. and Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology.*, 2:115–150.

[Fischer et al., 1985] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

[Foster et al., 2003] Foster, H., Uchitel, S., Magee, J., and Kramer, J. (2003). Model-based verification of Web service compositions. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering.*, ASE 2003, pages 152 – 161.

[Freund and Little, 2009] Freund, T. and Little, M. (2009). Web Services Business Activity (WS-BusinessActivity) Version 1.2. OASIS standard, OASIS. http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec-errata-os/wstx-wsba-1.1-spec-errata-os.html.

[Friedrich et al., 2010] Friedrich, G., Fugini, M., Mussi, E., Pernici, B., and Tagni, G. (2010). Exception Handling for Repair in Service-Based Processes. *IEEE Transactions on Software Engineering*, 36(2):198 –215.

[Fugini et al., 2008] Fugini, M. G., Pernici, B., and Ramoni, F. (2008). Quality analysis of composed services through fault injection. In *Proceedings of the 2007 International Conference on Business Process Management*, BPM 2007, pages 245–256, Berlin, Heidelberg. Springer-Verlag.

[Garcia-Molina, 1983] Garcia-Molina, H. (1983). Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8:186–213.

[Garcia-Molina and Salem, 1987] Garcia-Molina, H. and Salem, K. (1987). Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD 1987, pages 249–259, New York, NY, USA. ACM.

[Gardner and Harle, 1998] Gardner, R. and Harle, D. (1998). Pattern discovery and specification techniques for alarm correlation. In *Network Operations and Management Symposium, 1998. NOMS 98., IEEE*, volume 3, pages 713 –722 vol.3.

[Georgakopoulos et al., 1995] Georgakopoulos, D., Hornick, M., and Sheth, A. (1995). An overview of workflow management: from process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3:119–153.

[Goodenough, 1975] Goodenough, J. B. (1975). Exception handling: issues and a proposed notation. *Communications of the ACM*, 18:683–696.

[Gorbenko et al., 2007] Gorbenko, A., Kharchenko, V., and Romanovsky, A. (2007). On composing Dependable Web Services using undependendable web components. *International Journal of Simulation and Process Modelling*, 3(1/2):45–54.

[Gray, 1978] Gray, J. (1978). Notes on data base operating systems. In Bayer, R., Graham, R., and Seegmller, G., editors, *Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Berlin / Heidelberg.

[Green and Swets, 1966] Green, D. M. and Swets, J. A. (1966). *Signal Detection Theory and Psychophysics*. Wiley & Sons, Inc., 1st edition.

[Greenfield et al., 2003] Greenfield, P., Fekete, A., Jang, J., and Kuo, D. (2003). Compensation is Not Enough. In *Proceedings of the 7th International Conference on Enterprise Distributed Object Computing*, EDOC 2003, pages 232–, Washington, DC, USA. IEEE Computer Society.

[Guinea et al., 2011] Guinea, S., Kecskemeti, G., Marconi, A., and Wetzstein, B. (2011). Multi-layered monitoring and adaptation. In *Proceedings of the 9th International Conference on Service-Oriented Computing*, ICSOC 2011, pages 359–373, Berlin, Heidelberg. Springer-Verlag.

[Guo et al., 2006] Guo, Z., Jiang, G., Chen, H., and Yoshihira, K. (2006). Tracking Probabilistic Correlation of Monitoring Data for Fault Detection in Complex Systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN 2006, pages 259–268, Washington, DC, USA. IEEE Computer Society.

[Gupta et al., 2003] Gupta, M., Neogi, A., Agarwal, M., and Kar, G. (2003). Discovering Dynamic Dependencies in Enterprise Environments for Problem Determination. In Brunner, M. and Keller, A., editors, *Self-Managing Distributed Systems*, volume 2867 of *Lecture Notes in Computer Science*, pages 125–166. Springer Berlin / Heidelberg.

[Haerder and Reuter, 1983] Haerder, T. and Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317.

[Hagen and Alonso, 2000] Hagen, C. and Alonso, G. (2000). Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943 –958.

[Hand, 2009] Hand, D. J. (2009). Measuring classifier performance: a coherent alternative to the area under the ROC curve. *Machine Learning Journal*, 77(1):103–123.

[Hempstalk and Frank, 2008] Hempstalk, K. and Frank, E. (2008). Discriminating Against New Classes: One-Class versus Multi-Class Classification. In *Proceedings 21st Australasian Joint Conference on Artificial Intelligence*, JCAI 2008. Springer.

[Hempstalk et al., 2008] Hempstalk, K., Frank, E., and Witten, I. H. (2008). One-Class Classification by Combining Density and Class Probability Estimation. In *Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases - Part I*, ECML PKDD '08, pages 505–519, Berlin, Heidelberg. Springer-Verlag.

[Holden et al., 1993] Holden, J. R., Du, Z., and Ammon, H. L. (1993). Prediction of possible crystal structures for C-, H-, N-, O-, and F-containing organic compounds. *Journal of Computational Chemistry*, 14(4):422–437.

[Horwitz et al., 1990] Horwitz, S., Reps, T., and Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12:26–60.

[Huang et al., 1995] Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. (1995). Software rejuvenation: analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing - Volume III,'Highlights from Twenty-Five Years'.*, FTCS-25, pages 381 –390.

[Hughes, 1968] Hughes, G. (1968). On the mean accuracy of statistical pattern recognizers. *IEEE Transactions on Information Theory*, 14(1):55–63.

[Hyperic, 2012] Hyperic (2012). Hyperic SIGAR API. http://www.hyperic.com/products/sigar. accessed Mar 15, 2012.

[IBM, 2012] IBM (2012). WebSphere Application Server. http://www-01.ibm.com/software/webservers/appserv/was/. accessed Aug 23, 2012.

[ISO/IEC/(IEEE), 2011] ISO/IEC/(IEEE) (2011). ISO/IEC/IEEE 42010-2011 : Systems and software engineering - Architecture description. http://www.iso-architecture.org/42010/.

[Jordan and Evdemon, 2007] Jordan, D. and Evdemon, J. (2007). Web Services Business Process Execution Language (WS-BPEL) Version 2.0. Technical report, OASIS. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[Kar et al., 2000] Kar, G., Keller, A., and Calo, S. (2000). Managing application services over service provider networks: architecture and dependency analysis. In *Network Operations and Management Symposium, 2000. NOMS 2000. 2000 IEEE/IFIP*, pages 61 –74.

[Kashima et al., 2005] Kashima, H., Tsumura, T., Ide, T., Nogayama, T., Hirade, R., Etoh, H., and Fukuda, T. (2005). Network-Based Problem Detection for Distributed Systems. In *Proceedings of the*

21st International Conference on Data Engineering, ICDE 2005, pages 978–989, Washington, DC, USA. IEEE Computer Society.

[Katchabaw et al., 1997] Katchabaw, M., Howard, S., Lutfiyya, H., Marshall, A., and Bauer, M. (1997). Making distributed applications manageable through instrumentation. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 84–94.

[Kephart and Chess, 2003] Kephart, J. O. and Chess, D. M. (2003). The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50.

[Khalaf et al., 2009] Khalaf, R., Roller, D., and Leymann, F. (2009). Revisiting the Behavior of Fault and Compensation Handlers in WS-BPEL. In *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I*, OTM 2009, pages 286–303, Berlin, Heidelberg. Springer-Verlag.

[Kiciman and Fox, 2005] Kiciman, E. and Fox, A. (2005). Detecting application-level failures in component-based Internet services. *IEEE Transactions on Neural Networks*, 16(5):1027–1041.

[Klinger et al., 1995] Klinger, S., Yemini, S., Yemini, Y., Ohsie, D., and Stolfo, S. (1995). A coding approach to event correlation. In *Proceedings of the 4th International Symposium on Integrated Network Management IV*, IM 1995, pages 266–277, London, UK, UK. Chapman & Hall, Ltd.

[Knight and Leveson, 1986] Knight, J. C. and Leveson, N. G. (1986). An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1):96–109.

[Kulldorff, 1997] Kulldorff, M. (1997). A spatial scan statistic. *Communications in Statistics - Theory and Methods*, 26(6):1481–1496.

[Leaverton and Birch, 1969] Leaverton, P. and Birch, J. J. (1969). Small Sample Power Curves for the Two Sample Location Problem. *Technometrics*, 11(2):pp. 299–307.

[Lee et al., 2004] Lee, W., McGough, S., Newhouse, S., and Darlington, J. (2004). A Standard Based Approach to Job Submission through Web Services. In Cox, S., editor, *Proceedings of the 2004 UK e-Science All Hands Meeting, Nottingham*. UK EPSRC. ISBN 1-904425-21-6.

[Leners et al., 2011] Leners, J. B., Wu, H., Hung, W.-L., Aguilera, M. K., and Walfish, M. (2011). Detecting failures in distributed systems with the FALCON spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP 2011, pages 279–294. ACM.

[Lerner et al., 2010] Lerner, B., Christov, S., Osterweil, L., Bendraou, R., Kannengiesser, U., and Wise, A. (2010). Exception Handling Patterns for Process Modeling. *IEEE Transactions on Software Engineering*, 36(2):162 –183.

[Lim et al., 2008] Lim, C., Singh, N., and Yajnik, S. (2008). A log mining approach to failure analysis of enterprise telephony systems. In *Proceedings of the 2008 International Conference on Dependable Systems and Networks With FTCS and DCC*, DSN 2008, pages 398 –403. IEEE.

[Little and Wilkinson, 2009] Little, M. and Wilkinson, A. (2009). Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.2. OASIS standard, OASIS. http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-os/wstx-wsat-1.2-spec-os.html.

[Litzkow et al., 1988] Litzkow, M., Livny, M., and Mutka, M. (1988). Condor-a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, ICDCS 1988, pages 104–111. IEEE.

[Looker et al., 2007] Looker, N., Xu, J., and Munro, M. (2007). Determining the dependability of Service-Oriented Architectures. *International Journal of Simulation and Process Modelling*, 3(1/2):88 – 97.

[Lyu, 1996] Lyu, M. R. (1996). *Handbook of Software Reliability Engineering*. McGraw-Hill.

[Machine Learning Group at University of Waikato, 2012] Machine Learning Group at University of Waikato (2012). WEKA. http://www.cs.waikato.ac.nz/ml/weka/index.html. accessed Jul 12, 2012.

[Magee and Kramer, 1999] Magee, J. and Kramer, J. (1999). *Concurrency - State Models & Java Programs*. John Wiley & Sons.

[Magoutis et al., 2008] Magoutis, K., Devarakonda, M., Joukov, N., and Vogl, N. G. (2008). Galapagos: Model-Driven Discovery of End-to-End Application-Storage Relationships in Distributed Systems. *IBM Journal of Research and Development*, 52:367–377.

[Mann and Whitney, 1947] Mann, H. B. and Whitney, D. R. (1947). On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60.

[Maqbool and Babri, 2007] Maqbool, O. and Babri, H. (2007). Hierarchical Clustering for Software Architecture Recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780.

[Massie et al., 2004] Massie, M. L., Chun, B. N., and Culler, D. E. (2004). The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840.

[Mayer and Lübke, 2006] Mayer, P. and Lübke, D. (2006). Towards a BPEL unit testing framework. In *Proceedings of the 2006 workshop on Testing, Analysis, and Verification of Web Services and Applications*, TAV-WEB 2006, pages 33–42, New York, NY, USA. ACM.

[Mehrotra et al., 1992] Mehrotra, S., Rastogi, R., Silberschatz, A., and Korth, H. (1992). A transaction model for multidatabase systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, ICDCS 1992, pages 56 –63. IEEE.

[Microsoft, 2011] Microsoft (2011). Microsoft ṄET. http://www.microsoft.com/net/.

[Mills et al., 2010] Mills, D., Martin, J., Burbank, J., and Kasch, W. (2010). Network Time Protocol Version 4: Protocol and Algorithms Specification. Technical Report 5905, Internet Engineering Task Force. RFC 5905 (Standards Track).

[Mitra and Lafon, 2007] Mitra, N. and Lafon, Y. (2007). Simple Object Access Protocol (SOAP) Version 1.2 Part 0: Primer (Second Edition). Technical report, W3C. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[Mockapetric, 1987] Mockapetric, P. (1987). Domain Names - Concepts and Facilities. Technical Report 1034, Internet Engineering Task Force. RFC 1034 (Standards Track).

[Moore and McCabe, 2005] Moore, D. S. and McCabe, G. P. (2005). *Introduction to the practice of statistics*. W. H. Freeman and Co., New York, NY, USA, fifth edition.

[Moser et al., 2008] Moser, O., Rosenberg, F., and Dustdar, S. (2008). Non-intrusive monitoring and service adaptation for WS-BPEL. In *Proceedings of the 17th international conference on World Wide Web*, WWW 2008, pages 815–824, New York, NY, USA. ACM.

[Moser et al., 2010] Moser, O., Rosenberg, F., and Dustdar, S. (2010). Event driven monitoring for service composition infrastructures. In *Proceedings of the 11th International Conference on Web Information Systems Engineering*, WISE 2010, pages 38–51, Berlin, Heidelberg. Springer-Verlag.

[Moya and Hush, 1996] Moya, M. M. and Hush, D. R. (1996). Network constraints and multi-objective optimization for one-class classification. *Neural Networks*, 9(3):463 – 474.

[Moya et al., 1993] Moya, M. M., Koch, M. W., and Hostetler, L. D. (1993). One-class classifier networks for target recognition applications. *NASA STI/Recon Technical Report N*, 93:24043.

[Murray-Rust and Rzepa, 1999] Murray-Rust, P. and Rzepa, H. S. (1999). Chemical Markup, XML, and the Worldwide Web. 1. Basic Principles. *Journal of Chemical Information and Computer Sciences*, 39(6):928–942.

[Object Mangement Group (OMG), 2002] Object Mangement Group (OMG) (2002). OMG Unified Modeling Language (OMG UML), Superstructure, version 2.2. Technical report, Object Management Group.

[Open Services Gateway initiative Alliance, 2012] Open Services Gateway initiative Alliance (2012). Open Services Gateway initiative (OSGi. http://www.osgi.org/. accessed May 31, 2012.

[Oracle, 2011] Oracle (2011). Jave Platform, Enterprise Edition. http://www.oracle.com/technetwork/java/javaee/overview/index.html.

[Papazoglou, 2008] Papazoglou, M. P. (2008). *Web Services: Principles and Technology*. Prentice Hall.

[Patterson et al., 2002] Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., and Treuhaft, N. (2002). Recovery Oriented Computing (ROC): Motivation, Definition, Techniques. Technical report, Berkeley, CA, USA.

[Peltz, 2003] Peltz, C. (2003). Web services orchestration and choreography. *IEEE Computer*, 36(10):46 – 52.

[Plattner, 1984] Plattner, B. (1984). Real-Time Execution Monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756–764.

[Quinlan, 1993] Quinlan, J. R. (1993). *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Randell, 1975] Randell, B. (1975). System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, New York, NY, USA. ACM.

[Red Hat, Inc.,] Red Hat, Inc. Fedora Core. http://fedoraproject.org/.

[Red Hat, Inc., 2012] Red Hat, Inc. (2012). RHQ. http://www.jboss.org/rhq. accessed Mar 15, 2012.

[Russell and Norvig, 1995] Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1st edition.

[Salfner et al., 2010] Salfner, F., Lenk, M., and Malek, M. (2010). A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3):10:1–10:42.

[Scheffe, 1970] Scheffe, H. (1970). Practical Solutions of the Behrens-Fisher Problem. *Journal of the American Statistical Association*, 65(332):pp. 1501–1508.

[Schmerl et al., 2006] Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., and Yan, H. (2006). Discovering Architectures from Running Systems. *IEEE Transactions on Software Engineering*, 32(7):454–466.

[Skene et al., 2010] Skene, J., Raimondi, F., and Emmerich, W. (2010). Service-Level Agreements for Electronic Services. *IEEE Transactions on Software Engineering*, 36(2):288–304.

[Sneed and Huang, 2006] Sneed, H. M. and Huang, S. (2006). WSDLTest - A Tool for Testing Web Services. *IEEE International Workshop on Web Site Evolution*, 0:14–21.

[Splunk Inc., 2012] Splunk Inc. (2012). Splunk. http://www.splunk.com. accessed Jul 28, 2012.

[Stelling et al., 1998] Stelling, P., Foster, I. T., Kesselman, C., Lee, C. A., and von Laszewski, G. (1998). A Fault Detection Service for Wide Area Distributed Computations. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, HPDC 1998, pages 268–278, Washington, DC, USA. IEEE Computer Society.

[Sun Microsystems, 2006] Sun Microsystems, I. (2006). Java Management Extensions (JMX) Specification, version 1.4. Technical report. http://docs.oracle.com/javase/6/docs/technotes/guides/jmx/JMX_1_4_specification.pdf.

[Tax, 2001] Tax, D. M. J. (2001). *One-class classification*. PhD thesis, Delft University of Technology.

[The Apache Software Foundation, 2012a] The Apache Software Foundation (2012a). Apache Axis. http://ws.apache.org/axis/. accessed Mar 15, 2012.

[The Apache Software Foundation, 2012b] The Apache Software Foundation (2012b). Apache Tomcat. http://tomcat.apache.org/. accessed Apr 03, 2012.

[The University of Waikato, 2012] The University of Waikato (2012). weka.classifiers.meta.OneClassClassifiers. http://wiki.pentaho.com/display/DATAMINING/OneClassClassifier. accessed Jul 21, 2012.

[Tsai et al., 2003] Tsai, W., Paul, R., Cao, Z., Yu, L., and Saimi, A. (2003). Verification of Web services using an enhanced UDDI server. In *Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems*, WORDS 2003, pages 131–138. IEEE.

[Vaidyanathan and Trivedi, 1999] Vaidyanathan, K. and Trivedi, K. (1999). A measurement-based model for estimation of resource exhaustion in operational software systems. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, ISSRE 1999, pages 84–93. IEEE.

[Vilalta et al., 2002] Vilalta, R., Apte, C. V., Hellerstein, J. L., Ma, S., and Weiss, S. M. (2002). Predictive algorithms in the management of computer systems. *IBM Systems Journal*, 41(3):461 –474.

[Vogels, 1996] Vogels, W. (1996). World wide failures. In *Proceedings of the 7th Workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, EW 7, pages 115–120, New York, NY, USA. ACM.

[Vogels and Re, 2003] Vogels, W. and Re, C. (2003). WS-Membership - Failure Management in a Web-Services World. In *Proceedings of the 12th International World Wide Web Conference*, WWW 2003. International World Wide Web Conference Committee.

[Walmsley and Fallside, 2004] Walmsley, P. and Fallside, D. C. (2004). XML Schema Part 0: Primer Second Edition. W3C recommendation, W3C. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[Wang et al., 2011] Wang, C., Viswanathan, K., Lakshminarayan, C., Talwar, V., Satterfield, W., and Schwan, K. (2011). Statistical techniques for online anomaly detection in data centers. In *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management*, IM 2011, pages 385–392. IEEE.

[Wassermann, 2010] Wassermann, B. (2010). Improving wide-area distributed system availability. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE 2010, pages 347–348, New York, NY, USA. ACM.

[Wassermann and Emmerich, 2006] Wassermann, B. and Emmerich, W. (2006). Reliable Scientific Service Compositions. In Feuerlicht, G. and Zirpins, C., editors, *Proceedings of the 2nd International Workshop on Engineering Service-Oriented Applications: Design and Composition*, WESOA 2006, pages 14–25. Springer Verlag.

[Wassermann and Emmerich, 2011] Wassermann, B. and Emmerich, W. (2011). Monere: Monitoring of Service Compositions for Failure Diagnosis. In *Proceedings of the 9th International Conference on Service-Oriented Computing*, ICSOC 2011, pages 344–358, Berlin, Heidelberg. Springer-Verlag.

[Wassermann et al., 2009] Wassermann, B., Ludwig, H., Laredo, J., Bhattacharya, K., and Pasquale, L. (2009). Distributed Cross-Domain Change Management. In *Proceedings of 7th IEEE International Conference on Web Services*, ICWS 2009, pages 59–66. IEEE.

[Welch, 1947] Welch, B. L. (1947). The generalization of student's problem when several different population varlances are involved. *Biometrika*, 34(1-2):28–35.

[Wetzstein et al., 2009] Wetzstein, B., Leitner, P., Rosenberg, F., Brandic, I., Dustdar, S., and Leymann, F. (2009). Monitoring and Analyzing Influential Factors of Business Process Performance. In *Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference*, EDOC 2009, pages 141–150, Washington, DC, USA. IEEE Computer Society.

[Willock et al., 1995] Willock, D. J., Price, S. L., Leslie, M., and Catlow, C. R. A. (1995). The relaxation of molecular crystal structures using a distributed multipole electrostatic model. *Journal of Computational Chemistry*, 16(5):628–647.

[Witten and Frank, 2005] Witten, I. H. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2nd edition.

[Wohlin et al., 2000] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA.

[Xu et al., 1995] Xu, J., Randell, B., Romanovsky, A., Rubira, C., Stroud, R., and Wu, Z. (1995). Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing, Digest of Papers*, FTCS-25, pages 499–508. IEEE.

[Xu et al., 2005] Xu, W., Offutt, J., and Luo, J. (2005). Testing Web Services by XML Perturbation. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, ISSRE 2005, pages 257–266, Washington, DC, USA. IEEE Computer Society.

[Yahoo!, 2012] Yahoo! (2012). Yahoo Pipes. http://pipes.yahoo.com/pipes/. accessed Jun 07, 2012.

[Yan et al., 2006] Yan, J., Li, Z., Yuan, Y., Sun, W., and Zhang, J. (2006). BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, ISSRE 2006, pages 75–84. IEEE.

[Yan et al., 2005] Yan, Y., Pencole, Y., Cordier, M.-O., and Grastien, A. (2005). Monitoring Web service networks in a model-based approach. In *Proceedings of the 3rd IEEE European Conference on Web Services*, ECOWS 2005, pages 192–204. IEEE.

[Ye et al., 2009] Ye, C., Cheung, S.-C., Chan, W. K., and Xu, C. (2009). Atomicity Analysis of Service Composition across Organizations. *IEEE Transactions on Software Engineering*, 35(1):2–28.

[Zhang et al., 1994] Zhang, A., Nodine, M., Bhargava, B., and Bukhres, O. (1994). Ensuring relaxed atomicity for flexible transactions in multidatabase systems. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD 1994, pages 67–78, New York, NY, USA. ACM.

[Zhang and Zhang, 2005] Zhang, J. and Zhang, L.-J. (2005). Criteria analysis and validation of the reliability of Web services-oriented systems. In *Proceedings of the 3rd IEEE International Conference on Web Services*, ICWS 2005, pages 621–628. IEEE.

[Zhang et al., 2004] Zhang, M., Zhang, C., Pai, V., Peterson, L., and Wang, R. (2004). Planetseer: internet path failure monitoring and characterization in wide-area services. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, OSDI 2004, pages 12–12, Berkeley, CA, USA. USENIX Association.

[Zimmermann et al., 2012] Zimmermann, T., Nagappan, N., Guo, P. J., and Murphy, B. (2012). Characterizing and predicting which bugs get reopened. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1074–1083, Piscataway, NJ, USA. IEEE Press.