Dynamic, Performance-aware Configuration and Management of Web Service Monitoring Systems

Garth Heward

Submitted in fulfilment of the requirements of the degree of Doctor of Philosophy

2012
Abstract

The Service Oriented Architecture (SOA) paradigm provides for distributed software with loose coupling and a high level of business support. Web Services (WSs) are the dominant technology for implementing an SOA, and allow for business services to be offered through standardised web interfaces. The use of Web Services allows businesses to more effectively meet and measure business goals by mapping business services directly to Web Services, and business-level requirements to Service Level Agreements (SLAs).

Service consumers often have requirements on minimum acceptable quality levels for properties such as response time and availability. Therefore, service providers offer a guarantee of delivered Quality of Service (QoS) levels in the form of SLAs. The delivered quality levels defined in these SLAs are usually only available at run-time via monitoring. Thus, monitoring is used to demonstrate that services are meeting requirements at run-time. Since SLAs often have financial penalties for non-compliance, providing records of delivered QoS is valuable to Web Service providers.

Different architectures and monitoring requirements mandate different monitoring techniques, leading to systems of monitors of various types and capabilities being used in Web Services systems. Furthermore, whilst monitoring is beneficial, adding monitors to Web Services systems reduces delivered QoS. For example, a monitor acting as a firewall that stops messages and analyses them for security may increase the response time of services. This may in turn cause an SLA for response time to be breached, and penalties to be paid out to consumers. To simultaneously meet monitoring and QoS requirements, there is a need to carefully select which monitors should be used to observe which QoS aspects, and at what resolutions or frequencies. A solution that allows a service provider to configure their monitoring system in order to balance the coverage and QoS impacts of
monitoring would provide value to a service provider by minimising SLA violations whilst maximising monitoring coverage. This research is to investigate how to best achieve this monitoring coverage and QoS impacts trade-off.

This research has four major contributions: (1) an analysis and classification of web service monitors, including profiles of monitoring impacts; (2) a method to calculate optimal monitoring configurations at design-time based on monitoring benefits and costs; (3) a method to continuously re-optimise a monitoring system at run-time using feedback mechanisms and a heuristic optimisation algorithm; (4) a method to learn monitoring impacts by analysing monitoring logs.

Different monitor types are likely to have different impact profiles, based on their actions and locations. We have classified monitors according to their location and method of monitoring, and measured the impact profiles of each class of monitor. Understanding these impact profiles will help designers of Web Service monitoring systems to select monitors that minimise expensive impacts, and helps them to optimise monitoring configurations.

Based on the monitoring impact profiles, we have developed a technique that allows a service provider to balance delivered QoS with monitoring coverage. This includes a formal model of the monitoring optimisation problem, and a technique for solving the optimisation problem at design-time using a brute-force algorithm.

Extending design-time optimisation, we have developed a run-time optimisation framework that allows for a suite of monitors to be continuously optimised based on recently delivered QoS levels and current system requirements. This system includes a Genetic Algorithm (GA)-based heuristic optimisation routine that allows for large systems to be optimised quickly at run-time.

Finally, we increase the accuracy of run-time optimisation with a technique to learn monitoring impacts at run-time. This technique performs regression analysis on monitoring logs in order to assign measured monitoring impacts to each monitoring activity that has taken place.

Overall, this research provides the capability for Web Service providers to automatically optimise their WS monitoring system, allowing for requirements on monitoring to be met, whilst ensuring valuable SLAs are not breached. Case study experiments using our
approach have shown that the business value obtained from Web Services monitoring systems can be increased by 30% when optimisation is applied at design time, and a further 20% when optimisation is re-applied continuously at run-time.
Acknowledgements

This research was supported by the Australian Research Council in collaboration with CA Labs. Without this support, along with the generous support of Swinburne University of Technology, I could not have completed this thesis.

I would like to express my sincere gratitude to my supervisor, Professor Jun Han, who patiently guided me through all of the research involved for this thesis. Professor Han also provided invaluable feedback on publications throughout my candidature. I would also like to extend this thanks to my associate supervisors Professor Jean-Guy Schneider and Dr Ingo Müller from Swinburne University of Technology, and Dr Steven Versteeg from CA Technologies for their tireless feedback and mentoring throughout my candidature.

Thanks to my peers, especially Tan Phan, Cameron Hine and Aldeida Aleti for their feedback and advice.

I would like to acknowledge the support and encouragement of my parents and family. Most importantly, I would like to thank my wife Stephanie for her constant support and encouragement over the last four years, and for the sacrifices she has made to allow me to undertake this PhD.
Declaration

This is to certify that this thesis contains no material which has been accepted for the award of any other degree or diploma and that to the best of my knowledge this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Garth Heward
3 Web Service Monitoring Literature

3.1 An Introduction to Software Monitoring .......................... 22
3.2 Web Service Monitoring Requirements .......................... 23
3.3 Monitoring Individual Web Services .............................. 25
   3.3.1 Distributed Systems Manager ................................. 28
   3.3.2 RMCM .................................................. 29
   3.3.3 SLA Modelling and Monitoring .............................. 31
   3.3.4 Dynamic Service Monitoring Infrastructure ................. 32
   3.3.5 WSLA .................................................. 33
   3.3.6 CREMONA ............................................. 35
   3.3.7 Scalable SLA Monitors .................................... 36
   3.3.8 Privacy Agreement Monitoring .............................. 37
   3.3.9 Modelling and Monitoring Web Service Requirements ....... 38
   3.3.10 FSA-based Monitoring ................................... 40
   3.3.11 Timed Automata-based Monitoring ......................... 41
   3.3.12 Consumer-level Web Service Monitoring ................. 41
   3.3.13 SALMon .............................................. 41
   3.3.14 WSFL ................................................. 43
   3.3.15 SLA Monitorability ...................................... 45
3.4 Monitoring Service Compositions ................................. 45
   3.4.1 Control Point Selection for Monitoring .................... 49
   3.4.2 Optimal Web Service and Monitor Selection ............... 50
   3.4.3 Dynamic BPEL Monitoring ................................ 51
3.4.4 Self-monitoring BPEL ........................................... 53
3.4.5 LLAMA ......................................................... 54
3.4.6 FSM-based Web Service Behaviour Monitoring ............... 55
3.4.7 Web Service Composition Monitor ................................ 56
3.4.8 Automated SLA Monitoring Deployment ......................... 57
3.4.9 MDA-based Web Service Monitor Generation ................... 58
3.4.10 ML-Eh ......................................................... 60
3.4.11 Event Calculus-based Web Service Monitoring ............... 61
3.4.12 RTML ............................................................ 62
3.4.13 Monitoring BPEL Processes for Robustness .................... 63
3.4.14 UML-based Web Service Conversation Monitoring ........... 64
3.4.15 Monitoring BPEL Assertions .................................. 65
3.4.16 DEC-based Web Service Monitoring ............................ 67
3.4.17 Query-based Distributed Monitoring ............................ 67
3.4.18 SCALEA-G .................................................... 68
3.4.19 ECA-based Web Service Monitoring ............................ 70
3.4.20 WSLogA ....................................................... 72
3.5 Monitoring Resources for Scheduling ................................ 73
3.5.1 SLA-based Web Service Reservations ............................ 74
3.5.2 Resource Allocation for Web Services ............................ 75
3.6 Summary of Web Service Monitoring Literature ................... 76
3.6.1 Provide a Generic Classification of Monitors .................... 78
3.6.2 Calculate Net QoS Impacts of a Monitoring Configuration .... 78
3.6.3 Determine Business Value of a Monitoring Configuration ...... 79
3.6.4 Balance Monitoring Coverage with Monitoring Impacts to Maximise Business Value ........................................ 80
3.6.5 Maintain an Optimal Monitoring Configuration at Run-time .... 80
3.7 Open Research Issues ............................................. 81

4 Cost of Monitoring ................................................. 83
4.1 Monitor Classification .............................................. 85
4.1.1 Probe-based Monitors .......................................... 85
5 Design-Time Monitoring Optimisation

5.1 The Monitoring Optimisation Problem
5.2 A Model of the Monitoring Optimisation Problem
5.3 Optimisation Methodology
  5.3.1 Enumerator
  5.3.2 Impact Analyser
  5.3.3 Requirements Analyser
  5.3.4 Optimiser
5.4 iTravel Optimisation
  5.4.1 iTravel Implementation
  5.4.2 iTravel System Performance
  5.4.3 iTravel Monitoring Settings
  5.4.4 iTravel Requirements Analysis
6.3.4 Monitoring Managers ........................................ 168
6.3.5 QoS Analyser .................................................. 170
6.3.6 Summary ...................................................... 171
6.4 Heuristic Optimisation Method ................................. 172
   6.4.1 Heuristic Search Techniques .............................. 172
   6.4.2 Genetic Algorithm-based Solver .......................... 174
      6.4.2.1 Genetic Algorithm Parameters ...................... 175
6.5 Search Space Reduction ........................................ 176
   6.5.1 Search Space Pruner ...................................... 176
      6.5.1.1 Search Space Pruning Evaluation .................... 178
   6.5.2 Search Space Divider .................................... 180
      6.5.2.1 Search Space Division Evaluation .................. 183
6.6 Selection of GA Parameters ................................ 183
   6.6.1 Mutation Rate ............................................ 185
   6.6.2 Crossover Rate .......................................... 186
   6.6.3 Crossover Type .......................................... 186
   6.6.4 Population Size .......................................... 187
   6.6.5 Plateau Length .......................................... 189
      6.6.5.1 Genetic Memory Evaluation ......................... 189
   6.6.6 Summary .................................................. 192
6.7 Heuristic Optimisation Evaluation ............................ 192
   6.7.1 Methodology .............................................. 192
   6.7.2 GA Solver Solution Quality .............................. 193
      6.7.2.1 Pure Random Experiments ............................ 193
      6.7.2.2 Trended Random Experiments ......................... 193
   6.7.3 Performance .............................................. 198
   6.7.4 Scalability .............................................. 198
6.8 Effects of Run-Time Optimisation .............................. 202
6.9 Summary ...................................................... 202

7 Automated Monitoring Impact Analysis ......................... 205
   7.1 Analysis of Monitoring Impacts ............................. 206
7.1.1 Scenario .......................................................... 206
7.1.2 Problem Definition ............................................. 207
7.2 Impact Analysis Technique ....................................... 208
7.2.1 QoS Record Pre-processing .................................. 211
7.2.2 Impact Analysis ................................................ 213
    7.2.2.1 Null Impact ................................................ 213
    7.2.2.2 Monitoring Setting Impact .............................. 215
    7.2.2.3 Monitoring Overhead Impact ............................ 217
    7.2.2.4 Summary .................................................. 221
7.3 Experiments and Results ......................................... 222
    7.3.1 Prediction Error ............................................. 225
    7.3.2 Raw Error .................................................. 227
    7.3.3 Discussion .................................................. 227
7.4 Related Work .................................................... 229
7.5 Summary .......................................................... 230

8 Case Study .......................................................... 231
    8.1 System Description ............................................. 232
        8.1.1 System Structure ......................................... 232
        8.1.2 Requirements ............................................. 234
            8.1.2.1 SLA Requirements .................................... 238
            8.1.2.2 Public Consumer QoS Goals ......................... 238
            8.1.2.3 Business Requirements .............................. 239
            8.1.2.4 Legal Requirements ................................. 240
        8.1.3 Monitoring Capabilities .................................. 241
        8.1.4 Baseline QoS ............................................. 244
        8.1.5 Monitoring Impacts ....................................... 245
    8.2 Implementation ................................................ 252
        8.2.1 iTravel Services and Monitors .......................... 253
        8.2.2 Design-Time Optimisation Framework ................... 253
        8.2.3 Run-Time Optimisation Framework ....................... 254
    8.3 Initial Optimisation at Design Time ......................... 255
List of Figures

1.1 Web Services Publish-Find-Bind Architecture ........................................ 3
1.2 WS Monitoring Classification ................................................................. 5
2.1 iTravel System ..................................................................................... 16
3.1 RMCM Monitoring Framework [60] ......................................................... 30
3.2 SLA Interaction Diagram [27] ................................................................. 33
3.3 Cremona Architecture [34] ..................................................................... 35
3.4 SALmon Architecture [4] ..................................................................... 43
3.5 BPEL RTML Extension [7] ................................................................... 62
3.6 WS Conversation Monitoring [22] ......................................................... 65
3.7 Planning and Monitoring Framework [29] ............................................... 66
3.8 Quality of Service (QoS) Taxonomy [64] .................................................. 69
4.1 Classification of Monitoring Techniques .................................................. 86
4.2 Web Service Monitoring Scenarios ......................................................... 93
4.3 Effect of Proxy-based Interceptor on Response Time ............................... 97
4.4 5000ms Blocking Delay at Proxy-based Interceptor ............................... 99
4.5 CPU/RAM Intensive Processing at Interceptor on Server ....................... 100
4.6 Blocking and Non-blocking CPU/RAM Intensive Processing at Interceptor on Dedicated Node ................................................................. 101
4.7 Eavesdropping vs. No Eavesdropping .................................................... 103
5.1 Optimisation Entity Relationship Diagram ............................................. 110
5.2 Optimisation Framework ..................................................................... 113
5.3 iTravel System ................................................................................... 122
List of Tables

4.1 Node Specifications ................................................. 96
5.1 Monitoring Enumeration ........................................... 115
5.2 iTravel Monitoring Enumeration ................................. 124
5.3 Trended Random Search Space Parameters .................... 147
5.4 Trended Random Test Results .................................... 150
6.1 Divisible Search Space .............................................. 182
6.2 Trended Random Test Results .................................... 194
7.1 Raw QoS Records ................................................... 211
7.2 Processed QoS Records ............................................. 213
7.3 Experimental Scenario Impacts .................................. 223
8.1 Server Specifications ............................................... 232
8.2 iTravel Requirements ............................................... 235
8.3 iTravel Monitoring Capabilities ................................. 242
8.4 iTravel Monitoring Setting Impacts ............................. 246
8.5 iTravel Monitoring Overhead Impacts .......................... 252
8.6 iTravel Optimal Monitoring Configuration ..................... 257
8.7 iTravel Optimal and Max Monitoring Configuration QoS Levels .......................... 259
8.8 iTravel Case Study Run-Time Optimisation Reliability .......... 260
8.9 Response Time Impacts of Monitoring flight_info for Marketing Analysis in the iTravel Case Study ................................. 267
8.10 Response Time Impacts of Monitoring hotel_info for Marketing Analysis in the iTravel Case Study ........................... 268
8.11 Response Time Impacts of Monitoring book_flight for Marketing Analysis in the iTravel Case Study .................................................. 268
8.12 Response Time Impacts of Monitoring book_hotel for Marketing Analysis in the iTravel Case Study .................................................. 269
8.13 Reliability Impacts of Monitoring flight_info for Marketing Analysis in the iTravel Case Study .................................................. 270
8.14 Reliability Impacts of Monitoring hotel_info for Marketing Analysis in the iTravel Case Study .................................................. 271
8.15 Reliability Impacts of Monitoring book_flight for Marketing Analysis in the iTravel Case Study .................................................. 272
8.16 Reliability Impacts of Monitoring book_hotel for Marketing Analysis in the iTravel Case Study .................................................. 273
C.1 Web Service Monitoring Approaches .................................................. 312
Style Guide

Standardised formatting has been used throughout this Thesis. All acronyms are presented in their full form for the first use in each chapter, and their condensed form thereafter with hyperlinks to their definitions in the glossary.

- **System Elements** represent elements in a system such as specific services, monitors or quality types.

- **Research Components** represent system elements of our research, such as specific algorithms for optimisation or tools for performing experiments.

- **System Variables** represent variables in a system, such as those used in algorithms and formal definitions.
Chapter 1

Introduction

This chapter provides an introduction to Service Oriented Architecture (SOA) and Web Services (WSs) with emphasis on the benefits and costs of monitoring these systems. We define a set of research goals concerning Web Service monitoring that will be addressed by this thesis and the approach used for meeting these research goals.

Section 1 introduces SOAs and WSs as well as Service Level Agreements (SLAs). Section 1.2 introduces and describes the need for WS monitoring. Section 1.3 describes the impacts that WS monitoring may impose, and Sections 1.3.1 to 1.3.2 introduce the concepts of identifying WS monitoring impacts and balancing those impacts with monitoring coverage. Finally, Sections 1.4 to 1.6 describe our research goals, planned research approach, and research contributions.

1.1 Service-Oriented Architecture and Web Services

The growing SOA paradigm provides distributed software with loose coupling, user-level composition, and a high level of business support through aspects such as SLAs and various management standards[47]. Using an SOA, businesses define services, which are high-level software artefacts that directly represent business services. An example service may be one that provides airline customers the ability to search for a particular flight, with parameters such as date and time, departure and arrival locations, cost and airline, etc.

WSs represent the dominant means for implementation of an SOA. A set of eXtensible
Markup Language (XML) standards define Web Services technology\(^1\)\(^2\). Unlike traditional software, service consumers discover, bind to, and invoke Web Services over the Internet at run-time. Service providers can also compose (assemble) Web Services into more complex Web Services or service processes at run-time, using standards such as Web Services Business Process Execution Language (WS-BPEL).

Web Services allow service providers to offer business-level services through publicly available web interfaces. Web Services use a set of standards that allow for interoperability between services and service providers, composition of services, and additional attributes such as Quality of Service (QoS) and security to be defined for services in a uniform way.

These Web Service standards allow for any business to offer a business service through a stand-alone Web Service to a service consumer. Service consumers may be business entities within the company hosting the services, other businesses, public consumers, or service aggregators or resellers. The use of Web Services allows businesses to more effectively meet and measure business goals by mapping business services directly to Web Services, and business-level requirements to SLAs[47].

### 1.1.1 Web Services

As discussed above, WS-based systems are an implementation of an SOA. WS systems use a set of standards to promote software interoperability, by making business services offered over the Internet implementation-agnostic, with standardised interfaces.

Web Service Description Language (WSDL) is used to describe Web Services themselves[43]. Each Web Service in a system is described by a WSDL document, which describes aspects such as the functionality and interface of that service. Service providers make these WSDL documents available to potential service consumers, who may then use the information in the WSDL file to invoke the described Web Service.

Figure 1.1 shows the typical push-find-bind architecture of a Web Services system. Service Providers first Publish their Web Service’s WSDL description to a Service Broker (who may be the Service Provider). Service Consumers search for a Web Service that meets their requirements using the same Service Broker. When a Service Consumer discovers a service they wish to use, they use the service’s description from the Service Broker to Bind

\(^1\)\url{http://www.w3.org/2002/ws}\n\(^2\)\url{http://www.oasis-open.org/specs/}
to the service at the Service Provider. This action may be a simple network configuration, or involve more complex activities such as negotiating SLAs. After this, the Service Consumer may use that Service Provider’s service.

Using the architecture above, a service consumer may bind to Web Services at runtime and execute them as a required. Web Services may also be composed into more complex composite Web Services and business processes, described by standards such as WS-BPEL[42]. This process composition allows for complex business workflows to be directly encoded as a series of Web Service executions, with the ability to add rules on aspects such as transactions and security. For example, the set of web services “Provide Security ID”, “Provide Network Access” and “Initiate Payroll” may be composed into the service “Setup New Staff member”, allowing for unnecessary complexities of the business process to be hidden.

1.1.2 Service Level Agreements

Service consumers often have requirements on minimum acceptable quality levels. For example, a service consumer may require a minimum response time of 5 seconds in order to ensure usability of their system. Since consumers and providers of Web Services are typically separate businesses or business entities, service consumers may not be able to directly control services and service infrastructure in order to ensure these required QoS levels are met. Instead, service providers offer a guarantee of delivered QoS levels in the form of SLAs[27]. These SLAs often have a standardised form, such as Web Service Level Agreement (WSLA), allowing for automatic acceptance, rejection, and negotiation of service terms when binding to a new service[27].

SLAs are typically divided into a set of individual Service Level Objectives (SLOs). Each SLO provides one QoS target, such as “response time of the Purchase Book
service will be less than 5 seconds”. SLOs are often linked to penalties to be paid for then
the stated quality level is not met, and may be graded so that penalties increase as quality
decreases. SLAs may be negotiated individually between service providers and consumers,
or a provider may offer either a single SLA to all consumers, or a small set of SLAs for
different consumer classes such as gold, silver and bronze. We provide a survey of publicly
available SLAs in Appendix A.

1.2 Web Service Monitoring

Since Web Services may be dynamically bound and composed at run-time, their properties
(such as response time and correctness) are usually only available at run-time via moni-
toring and/or bind-time testing. Thus, monitoring Web Services increases confidence that
services are meeting requirements at run-time[8]. Monitoring of Web Services is required
in order for service consumers and service providers to measure various quality aspects
of a Web Service system. Service providers, Web Service middleware providers, service
consumers and other parties may be interested in both the functional and non-functional
properties of Web Services and their supporting infrastructure. Web Service monitoring is
also used to validate SLAs. Monitoring may be performed by a service provider, consumer,
or independent third party in order to measure a provider’s compliance with SLAs.

Service providers use the results of these monitoring mechanisms for the management
of Web Services and their supporting infrastructure. Web Service management provides
the ability to repair faults in a Web Service or supporting component in order to prevent
errors, or modify the Web Service’s supporting infrastructure in order to return a higher
quality of service to Web Service consumers. For this research, monitoring refers to the
measurement and initial analysis of information (e.g. measuring the response time of a
service invocation and determining if it is within acceptable levels). Management refers to
any actions on the Web Service system, such as replacing services within a composition,
re-composing a composition, or administration of underlying infrastructure such as routers
and servers.

There are numerous non-functional properties that one may wish to monitor and man-
age in a service-oriented system, such as performance (response time, resources consumed,
throughput), security (security model, trust in partners, certificate quality, and key qual-
ity), reliability, and availability [44, 45]. Different parties involved in using or providing Web Services are interested in different quality aspects. A provider of an advertiser-supported news portal based on Web Services may be interested in the response time of their services, and a provider of web 2.0 video streaming may be interested in availability and security. In these cases, run-time monitoring and management would increase confidence in the quality of the software, and allow for increased quality of software due to an increase in the number of faults that are detected and subsequently repaired.

Certain service properties such as response time can only be determined at run-time via monitoring[8]. Monitoring is generally performed by a monitoring agent, a component in a Web Service system that monitors some aspect of that system. For example, a monitoring agent may be a Web Service proxy-based interceptor that logs all messages that it sends and receives. Monitoring agents may also perform actions based on some initial measurement, such as blocking a message due to a security violation.

Web Service monitoring is also used to validate SLAs. Monitoring may be performed by a service provider, consumer, or independent third party in order to measure a provider’s compliance with SLAs. Since SLAs often have financial penalties for non-compliance, providing records of delivered QoS is important for Web Service providers. In fact, the value of monitoring a particular QoS can be directly linked to SLAs penalties.

We have developed a Web Service monitoring classification that divides monitors according to how they observe and interact with the monitored system, shown in Figure 1.2. Our classification divides monitors into Probes, Interceptors, and Eavesdroppers, shown

![Figure 1.2: WS Monitoring Classification](image-url)
with grey backgrounds in Figure 1.2. Probe-based monitors are those that make active requests from their target. For example, a probe may be a web service client that periodically executes a Web Service and validates the response using a test oracle, or measures QoS aspects such as response time. Interceptor-type monitors are those that intercept communications between services and their consumers. For example, an interceptor may be a proxy server through which all traffic is redirected, which logs all communications. Interceptors may also be software-based components such as software firewalls, which may analyse messages for QoS or security breaches, for example. Eavesdropper-based monitors passively intercept communications between service consumers and providers. For example, an eavesdropper may use a router to ‘tee’ (duplicate) packets to a server for logging and analysis. Our classification is designed to cover all levels of the communications stack, from hardware-level packet based monitoring, to software-level Simple Object Access Protocol (SOAP) message interception.

1.3 Impacts of Web Service Monitoring

Adding monitors to a Web Service system may impact that system’s delivered QoS, and this impact of monitors on the system should be identified and managed.

As an example impact, a monitor acting as a firewall that stops messages and analyses them for security may increase the response time of services. Monitors commonly impact performance attributes of services due to either extending the transmission path of communications via proxies, intercepting communications for analysis using firewalls, or performing detailed analysis of services or communications using the same, finite set of resources as the Web Services themselves.

From Figure 1.2, it can be seen that different monitor classes are likely to have different impact profiles, based on the actions they take and the locations of the monitors. Understanding these impact profiles would help designers of Web Service monitoring systems to select monitors that minimise expensive impacts, whilst providing adequate monitoring capabilities.
1.3.1 Identifying Monitoring Impacts

Just as the QoS of Web Services cannot always be established before run-time, the impact of Web Service monitors cannot always be measured before run-time. As such, it would be beneficial if the impacts of monitors were to be identified at run-time, as those monitors are used and as the system evolves. The impacts of monitors should be suitably modelled so that they are accurate enough to minimise error in predicted monitoring impacts.

One method to identify the impacts of monitors is to actively execute services with and without monitors enabled, and measure the difference in delivered quality levels. Whilst effective, this technique requires the ability to manually configure monitors for the sole purpose of measuring their impact. Since this requirement may not always be achievable, another method is required that measures run-time monitoring impacts without modifying the monitoring system.

This identifying of monitoring impacts without modifying the system may be achieved by performing regression analysis on monitoring logs, in order to assign measured monitoring impacts to each monitoring activity that has taken place.

1.3.2 Balancing Monitoring Impacts

As discussed above, Web Service providers are bound by requirements on delivered QoS that often require monitoring for validation. However, excessive monitoring may reduce delivered QoS and cause SLA violations. As such, there is a need to carefully select which monitors should be used to observe which QoS aspects in order to both meet requirements for observing delivered QoS and requirements on the level of that QoS.

A solution that allows a service provider to configure their monitoring system in order to balance delivered QoS and monitoring coverage would provide value to a service provider by minimising SLA violations and maximising monitoring coverage with available resources. Research has not discovered any systems that can achieve this monitoring and QoS trade-off.

1.3.3 Existing Research in Dynamic Web Service Monitoring

As identified above, there is a need to balance the costs and benefits of monitoring, from the perspective of a service provider. These costs should be able to directly reflect costs
from not meeting SLAs by either not performing monitoring, or not meeting a required QoS level.

Baresi and Guinea, 2005 and Baresi and Guinea, 2011 provide a mechanism to manually reduce monitoring coverage in generic Web Service systems and WS-BPEL processes, respectively. This mechanism gives each monitor in the system a threshold value, which denotes that monitor’s importance. A system administrator then sets a monitoring level, and all monitors with a threshold value above that monitoring level are enabled, whilst those monitors with a threshold value below that monitoring level are disabled. This allows for a simple, manual trade-off between monitoring coverage and monitoring’s performance impact. However, the system does not provide for automatic balancing of monitoring with delivered QoS, as neither system models the relationship between monitoring and its QoS impacts. Rather, this balancing is left to a system administrator to perform, in an ad-hoc way. Furthermore, the systems only allow for monitoring to be enabled or disabled at the whole-monitor level, rather than targeting individual services or quality types.

Bertolino et al., 2007 and Shao et al., 2010 each present a generic method for balancing monitoring coverage with performance at run-time. Shao et al., 2010 uses a framework for hot-swapping monitors as Java classes at run-time, in order to modify monitoring’s coverage and impact as system load and performance varies. However, this work does not consider costs or benefits from SLAs, as it is a generic monitoring model for cloud computing systems. Bertolino et al., 2007 does consider costs in terms of SLAs, and presents a generic technique for assigning more monitoring resources to those SLOs that are more likely to be breached. However, this is a high-level, generic approach rather than a complete solution. Furthermore, Bertolino et al., 2007 only considers monitoring costs in terms of resource consumption of monitors, rather than the direct impact that monitoring can have on services by performing actions such as redirecting messages for analysis.

1.4 Research Goals

None of the published work from our literature review in the fields of SOA and component-based software provides a method for complex monitoring and management of a Web Services system with a trade-off between monitoring cost and monitoring value. Such a
system would allow a service provider to monitor aspects of their Web Service system that they are valuable, and to vary the cost and coverage of the monitoring system according to the current value received. That is, they could choose to monitor only what they can afford to, or is good value to them at the time, based on current requirements. What they choose to monitor should be dynamic, so that it can be changed at run-time. In order to optimise monitoring, the costs of monitoring in terms of QoS impacts should be known. The web service provider should be able to automatically learn and update knowledge of these costs, as their system evolves at run-time.

The research questions for the above problems are:

1. How can the QoS impact of Web Service monitoring be modelled and accurately measured?
2. How can an optimal monitoring configuration be calculated and deployed at design time?
3. How can a management system repeatedly reconfigure service monitoring at run-time to maintain an optimal system state?
4. How can monitoring logs be used to measure and update knowledge on the impact of monitors at run-time?

1.5 Research Approach

The proposed solution to the problem outlined in the previous section is a dynamic and adaptive monitoring system specifically for Web Services. Although the focus of this research is specifically on Web Services, where possible the concepts used in the generation of this system have been generalised in order that they may be reused in other component-based software fields. The framework will give service providers a monitoring system with the following controls:

1. Which services to monitor;
2. At what level (e.g., interval/frequency) to monitor the selected services;
3. The method of monitoring; and
4. What QoS aspects to monitor.

The impact of the monitoring on the existing software system must be measurable, and the monitoring aspects of the system should be able to automatically tailor their behaviour based on the impact they are having on the system, and the current state of the system. To facilitate this, the system is built on a management framework that allows monitors to be configured at run-time. The proposed system provides the following functionality:

1. A method for measuring the impact of monitoring on the system;

2. A method for determining an optimal monitoring configuration based on requirements and SLAs;

3. A method for quickly re-calculating that optimal monitoring configuration at run-time, based on changes in system state and requirements; and

4. A management system that allows for these monitoring configurations to be applied to Web Service monitors;

The research involves quantitative verification of prototypes with direct performance comparisons, and analysis of algorithms for time/space complexity. The research is also validated with an extended case study.

The project is divided into the following major research components:

1. Literature Review;

2. Performance Impact Analysis;

3. Design-time Optimisation;

4. Run-time Optimisation;

5. Automated Impact Analysis; and

6. The iTravel Case Study.

For each of these research components, a detailed analysis of existing work was undertaken in order to determine if there exist any academic or commercial solutions that could be extended in order to achieve the desired functionality.
1.6 Research Contributions

As discussed, there exist gaps in current Web Service monitoring systems that if filled, would allow for:

- The ability to monitor Web Services with a known, and reduced performance impact;
- The ability to configure a suite of monitors to achieve maximum business value;
- The ability to use monitoring results at run-time to continuously re-configure monitors and maintain an optimal monitoring solution; and
- The ability to identify the impacts of monitors at run-time by analysing monitoring logs.

We have provided an introduction into Service Oriented Architecture and Web Services, which describes the purposes and benefits of each, as well as the burdens imposed by adopting WS-based systems. We have identified that WS-based systems require monitoring at run-time, in order to measure QoS and validate SLAs. Our research intends to measure the performance impacts of this monitoring and develop a method to minimise the costs of monitoring whilst maintaining acceptable monitoring coverage, both once-off at design-time, and continuously at run-time. We also leverage the results of monitoring at run-time in order to measure the impacts that WS-monitoring has on the monitored system. Our research is grounded in a formal model, and validated using theoretical analysis and implementations of case studies.

1.7 Thesis Structure

Chapter 2 introduces a motivating scenario that is used and extended throughout the thesis. This scenario is based on a Web Service provider, iTravel, who provides a set of services to travel agencies for booking flights and hotels.

Chapter 3 provides a review of Web Service monitoring literature, paying particular attention to those techniques that measure the impacts of monitoring, and balance them with monitoring coverage. Chapter 3 also references a survey of web service monitoring literature provided in Appendix C.
Chapter 4 describes the **Performance Impact Analysis** research, which involves research into the impact that Web Service monitoring has on monitored services. For example, the cost of probing a service versus the cost of a passive SOAP interceptor may be measured so that the user can decide which technique is more appropriate for them. These measurements can be as simple as measuring the average response time of a service with monitoring enabled and disabled. A literature review of performance impact analysis research has been performed, and a test bed and test cases for measurements have been created.

Chapter 5 describes the **Design-Time Optimisation** research, which involves modelling a Web Service monitoring system in terms of monitoring benefits and monitoring costs, and finding an optimal configuration for a set of monitors, according to those benefits and costs.

Chapter 6 describes the **Run-time Optimisation** research, which involves developing an optimisation algorithm that is fast enough to be executed at run-time, and a supporting management infrastructure that can be used to aggregate monitoring results and apply new monitoring configurations at run-time.

Chapter 7 describes **Automated Impact Analysis**, which extends the Run-time Optimisation system by using monitoring records to determine the impacts that each monitor has on each delivered QoS, through a form of regression analysis.

Chapter 8 describes an extension to the iTravel Case Study introduced in Chapter 2, providing both a motivating scenario and a validation mechanism for our research. For this case study, a complex Web Service system is described and implemented, and the effectiveness of our techniques is measured against the implemented system.
Chapter 2

Scenario

This chapter presents a detailed scenario for iTravel, a fictional provider of web services for the travel industry. The iTravel scenario has been designed to help highlight the problems identified by this research. Most importantly, iTravel includes a set of Quality of Service (QoS) and monitoring requirements that provide direct financial value. iTravel also has a suite of monitors of each class (Eavesdropper, Interceptor and Probe) that provide overlapping monitoring coverage. This allows for a wide range of monitoring configurations, making the selection of an optimal configuration challenging enough to merit an automated approach.

The iTravel scenario is used to demonstrate and measure the effectiveness of each of the proposed techniques in this Thesis. As such, the iTravel scenario is intended to be as realistic as possible whilst maintaining enough simplicity to communicate key concepts.

This research requires detailed measurement and analysis of systems, including aspects such as privacy protection and security compliance. Other information required includes business information such as consumer contracts and corporate policies. For these reasons, it was decided that the needs of communicating key research goals and results would be better met by a fictional business scenario than a business case study.

There are inherent weaknesses in using either a scenario or a case study. These are that a scenario or case study can only demonstrate potential benefits of any solution. These benefits may be indicative of results that can be expected from the solution, but this cannot be demonstrated without a large and detailed set of case studies. To increase confidence in the communicability of the techniques developed by this research, both the scenario and
randomly-generated problem sets have been applied to systems under investigation. These randomly-generated problem sets are intended to cover a wide enough range of parameters that a high level of confidence is achieved for both the communicability and likely benefits of the techniques developed.

Section 2.1 introduces the iTravel scenario, which is further justified in Section 2.2. Section 2.3 describes the iTravel system along with its typical performance levels and user requirements. Section 2.4 describes the impacts that monitoring has on the iTravel system. Section 2.5 describes planned future changes to the iTravel system. Sections 2.6 and 2.6 summarise the iTravel system and list the research requirements for the system, respectively.

2.1 iTravel Scenario

Consider a company, iTravel, offering Web Services to travel agencies for booking flights. iTravel has a Web Service flight_info, used to lookup flight information, and a service book_flight, used to book flights.

iTravel currently has 100 consumers who each pay a $100 monthly fee for access to the flight_info and book_flight services. In exchange for these account fees, iTravel provides the flight_info and book_flight services to their consumers with guarantees for response time, privacy, security compliance, and reliability. These guarantees are provided through Service Level Agreements (SLAs). If iTravel is unable to demonstrate that they have met these requirements from SLAs, they must refund affected consumers a percentage of their monthly account fees.

In addition to the set of consumer-sourced requirements, iTravel is bound by corporate governance-sourced requirements, relating to system security and consumer privacy. These requirements reflect local laws and business regulations, and iTravel is liable for large fines if they cannot demonstrate that these requirements have been met.

iTravel must not only meet the requirements detailed above, but also demonstrate that the requirements have been met. This demonstration is achieved with service monitoring. Over time, iTravel has progressively added monitors into its system, each meeting some new monitoring requirement. iTravel now has a suite of monitors which can measure response time and reliability, and detect security and privacy violations in their
In recent months, however, iTravel hasn’t met their SLA obligations for **response time** or **reliability**, and were required to refund the account fees of affected consumers. Investigation revealed that service monitors were imposing a large performance and quality burden on the Web Services system and the monitors in fact had been doing some overlapping monitoring. This overlapping monitoring means that multiple monitors were observing the same QoS through different channels. This increases the impact of monitoring without providing any extra QoS information, and is a sign that iTravel’s monitoring system has been poorly configured.

### 2.2 Justification of iTravel Scenario

The iTravel scenario represents a business offering only two Web Services, with five monitors and only four properties being measured by those monitors. Although the size of this scenario seems artificially limiting, the scale of the iTravel scenario has been selected for three main reasons. Firstly, there are enough parameters that optimisation of the system still has a measurable impact. Secondly, the system is small enough that it can be optimised using a brute-force approach, allowing for heuristic algorithms to be scored against a known, optimal baseline. Thirdly, it is expected that real-life systems will be divisible into smaller sub-systems, for which the techniques described here will be applied - i.e., it is not expected that optimisation would be applied to all of a company’s Web Services, but rather to each independent suite of related services. For these reasons, the seemingly limiting small size of the scenario is not overly unrealistic, and in fact helps to demonstrate the potential benefits of the techniques developed.

### 2.3 iTravel System

The iTravel scenario is depicted in Figure 2.1. As part of the normal application system, the Web Services are shown in light grey, rounded boxes in the figure; the Web Service consumers (there may be many) are shown in the light grey, square box; and the Web Service proxies are shown in boxes with dark grey backgrounds. The monitors are shown in numbered, square white boxes, including two Probe type monitors (**flight_info Probe**
and book_flight Probe), two Interceptor type monitors (book_flight Interceptor and flight_info Interceptor), and one Eavesdropper monitor (Generic Eavesdropper). Solid lines represent service invocation/communications, and the points where communications are intercepted by monitors are indicated by dashed lines.

The aspects of the services being monitored concern security, response time, privacy, and reliability. Security monitoring supports the assessment of iTravel’s compliance with security regulations by continuously producing well-defined and traceable evidence (hereafter simplified to ‘security’) by checking messages for known weaknesses such as Structured Query Language (SQL) injection and verifying that actions such as authentication have been performed. Response time monitoring measures the round-trip time for a service request, on the service provider’s side. Privacy monitoring supports the assessment of iTravel’s compliance with privacy regulations (hereafter simplified to ‘privacy’) by performing actions such as verifying and cleansing outgoing personal information. Monitoring for reliability checks that service requests receive a response, and that the response is properly formatted.

The probe monitors (flight_info Probe and book_flight Probe) are capable of meeting requirements for monitoring response time, reliability, security and privacy of the Web Services by invoking the services and pretending to be service consumers. The probes are hosted on the same server as the Web Services.

The interceptor monitors (flight_info Interceptor and book_flight Interceptor) measure the same properties, but do so by intercepting the service requests of consumers, rather than generating their own requests as probes do. The interceptors have the ability
to forward messages after analysing them, and therefore interceptors can filter or modify communications and perform functions such as firewalling. The interceptor monitors are hosted on their own, dedicated servers.

The eavesdropper monitor (Generic Eavesdropper) acts similarly to the interceptors; however, it cannot modify or stop a message from being transmitted, as it is only a passive listener. The Generic Eavesdropper is hosted on its own, dedicated server.

2.3.1 iTravel System Performance

The response time for both the flight_info and book_flight services in the iTravel scenario has been measured to be 2.5 seconds under ideal conditions of no other system load and no monitoring. All other aspects (security, reliability, privacy) are assumed 100% for our tests (i.e., we assume that the security, reliability, and privacy requirements of services are always achieved).

2.3.2 iTravel System Requirements

iTravel is bound by SLA and Corporate Governance (CG) requirements. The SLA requirements are from a single SLA that all service consumers have with iTravel, and the corporate governance requirements come from iTravel’s business and legal obligations. The requirements on iTravel are:

SLA1. flight_info response time will be \( \leq 10s \), penalty=33% account fee

SLA2. book_flight response time will be \( \leq 10s \), penalty=66% account fee

SLA3. flight_info will be \( \geq 95\% \) reliable, penalty=100% account fee

SLA4. book_flight will be \( \geq 98\% \) reliable, penalty=100% account fee

SLA5. flight_info will meet privacy requirements, penalty=100% account fee

SLA6. book_flight will meet privacy requirements, penalty=100% account fee

CG1. flight_info will meet security requirements, penalty=$10,000

CG2. book_flight will meet security requirements, penalty=$10,000
CG3. flight_info will meet privacy requirements, penalty=$5,000

CG4. book_flight will meet privacy requirements, penalty=$5,000

Each of the requirements in the above list describes a QoS threshold and penalty to be paid by iTravel when that threshold is not met.

The SLA-sourced requirements (SLA1–SLA6) describe requirements from SLAs. SLA1 and SLA2 describe requirements on response time for the flight_info and book_flight services, respectively. In both cases, if response time is over 10 seconds, iTravel must pay their consumers a penalty. Similarly, SLA3 and 4 describe requirements on reliability for each service, and SLA5 and SLA6 describe requirements on privacy. It should be noted that if more than one SLA-sourced requirement is not met, then service consumers may receive penalties above 100% of their account fees. For example, if SLA5 and SLA6 are both breached, consumers will receive a refund of 200% of their account fees. This type of penalty can be seen in public SLAs, such as the NetSuite SLA in Appendix A.

The Corporate Governance-sourced requirements (CG1–CG4) describe requirements from iTravel’s corporate governance and legal obligations. These are requirements on security (CG1) and CG2) and privacy (CG3 and CG4). The penalties for these requirements is paid to a body outside of iTravel (such as a parent agency or local government), but not directly to consumers. Furthermore, the penalties are fixed (e.g. the a flat $5,000 for CG3 and CG4), rather than per-consumer. This means that whilst individual penalties for Corporate Governance seem high, the penalties from SLAs may have a higher total cost if there are a large enough number of consumers.

2.4 iTravel Monitoring Impacts

As stated above, iTravel has not met the QoS aspects of their requirements in recent months. It has come to the attention of iTravel that their monitoring system is imposing a large burden on delivered QoS. Therefore, iTravel wishes to determine which monitors have impacts on which Qualities of Service, and the scale of those impacts. With this knowledge, iTravel will be able to select the most efficient of the available monitors1, and

---

1We assume that performing monitoring of the same quality of service with multiple monitors provides no extra benefit
disable monitors when their impact is too large.

2.5 iTravel Future Changes

In coming months, iTravel intends to open their system to the public as an advertiser-supported platform. This will introduce a new set of requirements and burdens on the iTravel system, including changes to user load, system architecture, monitors, and requirements. iTravel will require knowledge of monitoring impacts in this new system. However, attributes of this system including user load, monitors and requirements are expected to change at run-time, and so the impacts of monitors need to be learned at run-time, and the monitors will need to be re-configured accordingly.

2.6 iTravel Scenario Summary

The iTravel scenario represents a medium-sized business scenario with complex monitoring and QoS requirements. This scenario is used (with appropriate extensions and modifications) throughout this Thesis in order to communicate key problems and our solutions to them, and demonstrate the effectiveness of those solutions.

2.7 Requirements for Monitoring and Monitoring Management

Based on the scenario described above, iTravel requires a mechanism to configure their Web Service monitoring system in order to observe as many important Qualities of Service as possible, without causing unnecessary burdens on that delivered QoS. The goals for iTravel align with our research problems identified in Chapter 1. iTravel has the following requirements:

1. A generic classification of monitors and monitoring impacts, allowing for trade-offs between monitoring capabilities and their impacts

2. A method to calculate an optimal monitoring configuration and apply it at design time, including:
(a) A method to calculate net QoS impacts of a given monitoring configuration

(b) A method to determine net business value associated with a given monitoring configuration

(c) A method to optimise a monitoring configuration by balancing monitoring coverage with monitoring’s QoS impacts

3. A method to repeatedly reconfigure service monitoring at run-time to maintain an optimal system state

4. A method to update knowledge on the impact of monitors at run-time using information from monitoring logs.

These requirements are used to discuss work related to web services monitoring in Chapter 3 and work related to impact analysis in Chapter 7.

2.8 Summary

We have presented a scenario for iTravel, a provider of travel-related web services. iTravel has strict monitoring and QoS requirements, which must be met at run-time in order to maximise business value. This scenario is used to motivate and communicate our key research requirements and contributions throughout the rest of the Thesis, and is extended into a case study evaluation in Chapter 8.
Chapter 3

Web Service Monitoring

Literature

This chapter provides a literature review covering existing software monitoring research, focusing on monitoring distributed systems, and dynamic and adaptive monitoring solutions. The discussion is divided into work on individual Web Service (WS) monitoring, Web Service process monitoring, Service Level Agreement (SLA) monitoring, and resource monitoring.

Throughout this chapter, we focus on monitoring approaches that are related to Web Service-based and distributed systems. The monitoring approaches are classified and reviewed according to properties we consider important, which are monitoring’s performance impacts, monitoring capabilities, and the monitoring system’s flexibility. We also identify which of the approaches reviewed meet which requirements identified in Chapter 2.

Section 3.1 introduces software monitoring. Section 3.2 introduces WS monitoring requirements. Sections 3.3, 3.4 and 3.5 survey existing literature on monitoring individual Web Services, monitoring Web Service compositions and monitoring system resources for scheduling purposes, respectively. We provide a summary of our literature survey in Section 3.5.2 and list the set of remaining research problems in Section 3.7.
3.1 An Introduction to Software Monitoring

The continuous increase of complexity and criticality of software systems means that verification and validation of software must often be extended beyond design-time and development-time testing. Run-time monitoring of software systems allows for the functional properties of those systems to be continuously or periodically validated at run-time. For example, a monitor may observe the interactions of a file server in order to ensure that transmitted files are structurally valid. Run-time monitoring is also used to measure non-functional properties of software. These properties include performance-related attributes such as response time and throughput, and other quality types such as availability, reliability, and security. For example, a monitor may observe the interactions of a file server in order to ensure that security rules (e.g. that a user is not performing actions they do not have permission for) have not been violated.

As discussed in Chapter 1, software systems based on Web Services allow for a business to offer services through a uniform software interface over the Internet. The functional and non-functional properties of Web Services will be of interest to both the business providing a service and consumers of that service. As with traditional software systems, monitoring of WS-based software helps to increase confidence that various properties have been met at run-time. However, WS-based software systems introduce further requirements for monitoring. Firstly, services are often bound by SLAs. These are contracts between the service provider and service consumer that define the quality levels that must be achieved for a service or set of services. SLAs may include penalties to be paid from the service provider to the service consumer if certain quality level goals are not met [27]. Secondly, service providers may be bound by corporate governance or legal policies that require properties such as those concerning security and privacy to be met and validated through monitoring. Thirdly, certain Quality of Service (QoS) and functional properties of services may not be able to be validated until run-time due to the dynamic nature of Web Services. This dynamic nature is due to activities such as services being composed at run-time, or the resources available to services being modified at run-time, in addition to long-term variations in user load. Monitoring provides an effective mechanism for establishing the QoS and functional correctness of these services during and after execution.

Software monitoring can be achieved with various techniques, and at different abstrac-
tion levels. For example, monitoring may be achieved with hardware-based monitors that directly observe process execution [65], proxy servers that observe network traffic [28], monitoring code built into middleware [20], or the use of aspect-oriented programming to dynamically enable and disable monitoring code at run-time [19]. Software may also be monitored with probes by executing the software and comparing the output with a test oracle [38].

For our purposes, WS monitoring is defined as measuring or detecting and recording any aspects of either individual services, service compositions, or other aspects of an implemented WS-based system. Although the line between monitoring and managing is drawn at whether the component modifies the service or system, there are some management aspects covered in this research, since management and monitoring are closely linked due to most management systems performing or relying on monitoring.

3.2 Web Service Monitoring Requirements

In an environment where resources are constrained, it is not possible to monitor all aspects of all services all of the time. In this case, some intelligence should be used in order to achieve the best results of monitoring with limited resources. This concept is similar to testing software. It is not possible to test software at every state it could ever exist in, so a tester must maximise efficiency by choosing test cases that provide the greatest coverage with the least effort, and spend more resources testing those pieces of code that are either more important or more likely to fail.

Assuming that there are limited resources, those resources should be spent getting the greatest value from monitoring possible. Instead of spending 50% of monitoring resources on an established, mature service and 50% on a brand new service, an administrator should be able to allocate more monitoring resources to the new service. Similarly, there should be a system in place which is able to increase or reduce monitoring based on the load on services. Increasing or reducing monitoring may mean modifying the parameters that are monitored, changing the frequency, or taking measures such as not probing a service, and just passively monitor the service’s interactions instead. This is only applicable where monitoring is not at a level greater than that mandated.

‘Better monitoring’ is defined as monitoring that takes relatively less resources to ex-
ecute and detects more important behaviours of the Service Oriented Architecture (SOA) system than a naïve ‘monitor everything’ approach. There are a handful of techniques that a designer of a WS monitoring system can use in order to achieve better monitoring, and these include:

- Carefully designing how the monitors interact with the WS-based system. For example, a monitor that passively intercepts messages between Web Services should have less impact than a monitor that probes Web Services for a response every time periods;

- Designing the monitors to modify their behaviour at run-time based on the results of their monitoring, which allows a monitor to focus on services that may currently be under higher load, or showing signs of failing;

- Providing the ability for the administrator of the WS-based system to tailor various aspects of the monitoring system at run-time in order to focus on specific areas without causing too much of a performance impact. The tailoring could include the ability to simply turn monitoring on or off for specific services, or it could be as complex as having levels of monitoring from 1 to 100 for each monitorable aspect of the system. For example, an administrator may be concerned with a single element from an SLA, or a single aspect of the entire system such as memory usage or throughput; and

- Making the monitoring proactive, by configuring it to monitor the system in a way that will not cause breaches in important requirements whilst not under-monitoring when it is undesirable to do so.

All of these aspects of service monitoring are critical to the effectiveness and practicality of an SOA monitoring system.

As discussed in Chapters 1 and 2, it is important that the benefits of any monitoring solution outweigh its costs. This can be achieved by minimising the costs of monitoring (e.g. by using efficient monitors and remotely analysing monitoring data), reducing the frequency or resolution of monitoring (e.g. by sampling instead of monitoring every transaction), or disabling some monitoring altogether. We have generalised the set of requirements identified in Chapter 2 into a set of research requirements:
1. A generic classification of monitors and monitoring impacts;

2. The ability to perform dynamic modification on monitoring configurations at design-time and run-time;

3. The ability to perform impact trade-off by balancing monitoring coverage with monitoring impacts, which can be considered in two stages:
   (a) Complete coverage solutions, which find the optimal monitoring configuration for complete monitoring coverage; and
   (b) Incomplete coverage solutions, which find the optimal monitoring configuration and may include incomplete monitoring coverage.

4. The ability to calculate the net business cost of monitoring impacts by linking monitoring’s QoS impacts directly to costs that will be incurred from SLAs or other business requirements;

5. A run-time monitoring management framework; and

6. A method to measure monitoring impacts on a run-time system without having to actively modify that system.

We have identified which of the above requirements has been met by each of the works reviewed in this chapter, with the exceptions of requirement 5, which is a specific solution for our optimisation technique, and requirement 6, for which related work is discussed in Chapter 7.

### 3.3 Monitoring Individual Web Services

This section reviews techniques for monitoring individual Web Services, grid and cloud applications. These techniques do not specifically consider the interactions within service processes. However, monitoring of individual services is still useful if the properties of those services are important. Note that the services monitored by these solutions are not necessarily atomic (they may represent a service process), however service processes are treated as black boxes, and monitored in the same way as atomic services.
Abdu et al., 2002 and Shao et al., 2010 present systems that try to minimise monitoring impact whilst monitoring individual services. Abdu et al., 2002 presents a method for creating a monitoring solution for a set of distributed services. This solution is intended to minimise the use of a particular resource, such as bandwidth or processor usage. Shao et al., 2010 presents a framework for monitoring services and resources in a cloud-based system, such that the monitoring solution adapts as the monitored system changes at run-time.

Comuzzi et al., 2009 and Skene et al., 2007 present methods for evaluating the monitorability of SLAs when accepting consumer requests. Comuzzi et al., 2009 presents a technique for designing and monitoring SLAs in WS-based systems. SLAs are modelled hierarchically, and mirror the structure of WS compositions. Historical monitoring results are used to ensure Service Level Objectives (SLOs) are not offered that are unlikely to be met. Comuzzi and Spanoudakis, 2010 presents a solution for managing a suite of monitors for SLA verification, but differs from the authors previous work in that SLAs are only accepted, rather than being created, and the acceptance is based on whether or not each SLO of the new SLA is deemed ‘monitorable’. The approach includes a framework for dynamically configuring monitors based on what SLOs need to be observed, using a common interface that describes observational and analytical capabilities of available monitors. Skene et al., 2007 presents an evaluation of the monitorability of SLAs in application service provision scenarios. The authors argue that if the targets of an SLA are not able to be monitored effectively, then the SLA is not likely to be able to be enforced.

Keller and Ludwig, 2002 and Ludwig et al., 2004 present a system for defining SLAs using WSLA, and monitoring them with a custom middleware solution. Ludwig et al., 2004 differs from Keller and Ludwig, 2002 primarily in that in Ludwig et al., 2004 the service provider will accept and guarantee SLAs over time, based on current loads and incoming requests. Keller and Ludwig, 2002 presents an IBM SLA definition and monitoring framework (Web Service Level Agreement (WSLA)) for inter-domain services. The framework allows for collaborative SLA definition between the service consumer and provider. The SLA is appended to the target service’s Web Service Description Language (WSDL) description. The framework provides services for monitoring, reporting and management.
Ludwig et al., 2004 presents an architecture (Cremona) for dynamic creation and monitoring of WS-Agreement SLAs for Web Services. Cremona is a middleware containing a Java library that provides implementations for the required WS-Agreement interfaces, management for agreements, and service environment abstractions to simplify implementation. The architecture is aimed at customised QoS guarantees, based on agreements that include consumer input. The aim is to have SLAs that define the QoS guarantee for particular time periods and usage patterns. The authors propose to extend the SOA publish-find-bind model to include these agreements. Since the system is designed to run below the business level, it only supports technical agreements, e.g. throughput, and cannot support relationships between agreements, complex agreements (multi-party, etc), or reasoning based on composition.

Bertolino et al., 2007 presents a system that assigns more monitoring resources to those SLOs that are close to being breached, in order to reduce SLA violations.

Benbernou et al., 2007, Robinson, 2003, Li et al., 2006, and Raimondi et al., 2007 present systems for monitoring Web Services by modelling requirements as formal models, and validating run-time events against those models. Benbernou et al., 2007 presents a method for monitoring the compliance of privacy requirements on Web Services by comparing run-time events to a pre-defined state machine of privacy requirements. Robinson, 2003 presents an abstract framework for monitoring SLA-based requirements in SOAs by modelling requirements in KAOS and mapping that to a set of monitors, which monitor both Web Services interactions for meeting requirements, and any events that are known to cause requirements not to be met. Li et al., 2006 presents a framework for validating service interactions by monitoring and validating the run time interaction behaviour of services. The framework models service interaction constraints declaratively with Finite State Automata (FSAs), and verifies those interactions by intercepting messages between services. Raimondi et al., 2007 presents a method for using timed automata to verify service conversations, by analysing time-stamped Simple Object Access Protocol (SOAP) logs of those conversations.

Brenner et al., 2007 presents a method for using test probes to verify functional correctness of services.

Ameller and Franch, 2008 present a framework for replacing or re-composing ser-
vices when SLAs may be breached. Monitoring of SOA-based systems is performed in order to validate SLA requirements. The framework allows for monitors of any type to be used to observe services, with monitoring results stored in a central database. These results are analysed for compliance to SLAs.

Sahai et al., 2002 considers SLA measurement from the consumer’s perspective, and presents a method for automating the creation, monitoring and management of SLAs. SLAs are composed of SLOs, each of which is linked to specific monitoring metrics. Monitoring is based on SOAP interceptors, distributed across provider’s and consumer’s sites.

3.3.1 Distributed Systems Manager

Abdu et al., 2002 identifies the need to optimise the management of distributed systems components based on system and business requirements. For this work, management agents perform both monitoring and management of components in a distributed system. This may include WS-based services as managed components, and WS monitors and custom Managers as the management agents.

The requirements for monitoring are specified in a Management Policy, which also contains a description of monitoring capabilities. When optimising, the authors consider the type and location of monitors in a system, as well as agents that perform analysis. For example, there may be a choice between having a monitor measuring response time of a service and calculating the average, or having that monitor report its results to an aggregator on another server which averages the results. The authors note that there exist trade-offs between resource types used, as having a local monitor will reduce communication usage, but increase CPU and RAM usage on the same machine as the monitored component. However, their optimisation system optimises for only one resource type. Furthermore, the authors assume that all costs are provided to the system. The time for a monitoring result to be returned is also considered, so that there is a trade-off between a target resource type’s consumption and monitoring timeliness, for each monitored component. Whilst identifying that the management system must be capable of staying optimal through reconfiguration as the system and its requirements change, the authors have not described a mechanism to achieve this. The monitoring optimisation problem
is modelled such that each (monitor, location, component) tuple is assigned 1 or 0, depending on if that monitor observes that component from that location, or not. Then, optimisation consists of finding the complete set of (monitor, location, component) tuples that meet requirements, whilst minimising the resource of interest (such as bandwidth). Here, requirements are specific constraints such as stating a service must be monitored at all time, or that for a monitor to be used, it must observe at least two components. This optimisation problem is modelled as a binary Integer Linear Programming (ILP) problem.

The authors have implemented their technique and measured its effectiveness in terms of resource consumption, when compared to three basic configurations. This experiment was optimising bandwidth usage, and the measured usage of the optimal configuration was approximately twice as low as the next best model measured.

Whilst this work optimises a monitoring configuration in a distributed system according to the cost of that monitoring, it does not meet our requirements in three key respects. Firstly, it only considers one resource type as an optimisation goal. As such, it cannot balance resource usage types, or optimise according to each QoS requirement. Secondly, this work searches for a monitoring solution with complete coverage, rather than selecting only that monitoring which is valuable. Thirdly, optimisation is performed based on resource types rather than delivered QoS. This means that requirements on QoS cannot be used to find an optimal monitoring solution, in terms of net business utility.

3.3.2 RMCM

The authors identify the need to monitor in cloud-based systems in order to measure resource consumption, functional correctness and QoS, and learn patterns of service use [60]. The authors propose a Runtime Model for Cloud Monitoring (RMCM) to define abstract monitoring solutions. This RMCM is implemented in a monitoring framework, with a goal to balance the benefits and costs of monitoring. The flexibility of monitoring is seen as particularly important in a cloud-based system, in order to ensure that monitoring scales with elastic computing resources and provides information for resource management.

The model-based approach presented allows for a generic monitoring architecture to be instantiated at various locations, such as the cloud operator, service provider and service consumer. The model allows for low-level monitoring data to be composed into higher-
level information relevant to system administrators. The model is designed to reflect the run-time system, and change with that system. To achieve this, the model provides a fixed set of cloud entities in a hierarchy, including entities such as operating systems, physical machines, applications, and service consumers.

The authors identify the need for the overhead impact of monitoring to be carefully managed, in order to ensure cloud performance is not overly impacted. This is achieved through a flexible monitoring architecture with trade-offs between run-time overhead and monitoring capability.

Figure 3.1: RMCM Monitoring Framework [60]

Figure 3.1 shows the proposed monitoring framework based on the RMCM model. A monitoring agent is deployed on each Virtual Machine (VM) in the system, and processes all of that virtual machine’s monitoring data. There is a fixed set of possible monitors available on each VM, the ‘Monitoring Facilities’ in Figure 3.1. Each of these monitoring types represents a different monitoring capability and has a different impact profile. For this reason, the authors attempt to balance the monitoring coverage provided against the run-time overhead of monitoring.

The authors have implemented all monitors using Java, and run-time balancing of monitoring overhead is achieved through hot-swapping of Java classes during execution. This is performed manually by a human, although the authors note that automatic management is the goal of their work.

Whilst the proposed framework has been prototyped and implemented on a case study system, this was a proof-of-concept, and performance testing has not been performed. The authors note the need to measure the performance impact of monitoring at run-time, and
propose to analyse performance records to determine the impact of various monitoring configurations.

Whilst identifying the need to trade-off between monitoring capabilities and the performance impact of monitoring, this work is targeted at a generic model for cloud computing, and as such does not consider SLAs as a source of benefit or cost for monitoring. Furthermore, the costs of monitoring are manually provided, and relate to performance impacts alone. Similarly, benefits of monitoring are manually provided. There is no automatic optimisation of monitoring at either design-time or run-time, although authors identify this as potential future work.

3.3.3 SLA Modelling and Monitoring

The authors identify the need for the establishment and monitoring of SLAs in WS systems [18]. These SLAs are modelled as hierarchies which mirror composite service structures and allow for SLAs to be composed and aggregated. SLAs are negotiated considering multiple stakeholders, and are designed to be validated with service monitoring.

SLAs are negotiated between the provider and each consumer with a standard process of a proposal followed by repeated counter-proposals and bargaining, and ended by an acceptance or rejection of the SLA terms. Whilst the negotiation procedures for each party are not detailed, service providers may use historical monitoring data to ensure that new SLAs are likely to be met. For example, if a service has an average response time of 10 seconds, a provider would not agree to an SLA that had a response time requirement of 5 seconds. Furthermore, the authors identify the need to assess the ‘monitorability’ of qualities of service in SLAs. If it is not possible to monitor some QoS aspect in an SLA offer, then that offer is rejected by the service provider.

A publish/subscribe based event bus uses Event Captors to capture data from service executions, which is provided to monitors for analysis and verification against SLAs. Event Captor is a generic term for a component that observes some WS component. Proxies that intercept messages and measure response time, or network-based eavesdroppers logging reliability are both examples of Event Captors. The set of Event Captors available is used to generate the set of Monitoring Capabilities for the system.

The authors do not consider the performance impacts of monitoring, and only consider
changing monitoring configurations in order to meet new SLAs. Furthermore, whilst the system has been prototyped, no performance evaluation has been performed.

3.3.4 Dynamic Service Monitoring Infrastructure

The authors identify the need for WS monitoring systems to be dynamically updated at run-time, as the system and SLAs imposed on it evolve [17]. To meet this need, the authors propose an approach to establish and monitor SLAs, which determines the monitorability of SLAs and configures monitors based on current SLAs. The approach has been designed so that either a service consumer or service provider may use the monitoring management system.

All monitors in the target system are provided with common interfaces, defined with an eXtensible Markup Language (XML) schema. These interfaces allow for a management agent to gather information from the monitor on what SLOs it is capable of observing, and allow for an analysis agent to gather monitoring results for SLA validation.

When a new SLA is received, a manager searches for a set of monitors that can observe all SLOs in that SLA. If a set of monitors that can provide complete coverage is found, then the SLA is accepted and the monitors are configured to observe those SLOs accordingly. Similarly, if system monitors are modified, then the management agent will search for replacements in order to ensure complete monitoring coverage is maintained.

The system divides monitors into two classes: internal event-based monitors such as those that may observe SOAP server logs and exceptions, and external monitors, which may be of any type, but require an extra interface definition and Uniform Resource Identifier (URI).

The system has been prototyped, and measured for configuration performance on a small case study. That is, the time taken to calculate and apply an appropriate monitoring configuration based on a new SLA was measured. Neither the actual monitoring performance or the performance of the system at run-time were measured.

The authors have identified that selecting the ‘best’ monitor from a set of functionally equivalent monitors is important future work. It is unclear what ‘best’ is, in this case. Whilst this work allows for dynamic selection of monitors at run-time, the work searches for a monitoring solution with complete coverage, rather than trading off between monitoring...
coverage and cost.

### 3.3.5 WSLA

Keller and Ludwig, 2002 presents an IBM SLA definition and monitoring framework (WSLA) for inter-domain and inter-business services. The framework allows for automated definition, negotiation, deployment, monitoring and enforcement of SLAs.

The framework divides SLA metrics into Resource Metrics (basic raw elements such as time to respond for an individual service invocation), Composite Metrics (aggregation of Resource and or Composite metrics as well as filters, e.g. top 10), SLA Parameters (filtered Resource or Composite metrics for a particular customer, providing high and low watermarks), and Business Metrics (customer defined mappings of SLA parameters to financial terms which are private to the customer). The SLA interaction diagram is shown in Figure 3.2.

![Figure 3.2: SLA Interaction Diagram [27]](image)

The SLA in the framework extends the WSDL description of the target WS. The phases in Figure 3.2 are enacted as follows:

1. SLA Negotiation and Establishment, performed by means of an SLA Establishment Service, which provides authoring capabilities to the service customer and provider.
The SLA Establishment Service provides tools for aggregating SLA parameters, gaining both parties’ approval, adding secondary parties, and publishing the SLA.

2. SLA Deployment, performed by an SLA Deployment Service, which verifies the validity of the SLA, and distributes it to the Measurement, Condition Evaluation, Management, and Business Entity components as appropriate. The signatory parties must define which elements of the SLA should be distributed to which components and parties.

3. Measurement and Reporting, which configures the system for measurement and retrieval of resource metrics required by the SLA. A Measurement Service is used to measure system-level SLA parameters either directly (from managed resources) or indirectly (by probing the service provider or intercepting messages). There may be multiple Measurement Services each measuring different SLA aspects. A Condition Evaluation Service is used to compare current SLA parameter values against SLA defined guarantees, and notify the management system of any breaches.

4. Corrective Management Actions, which are provided by a Management Service and Business Entity. The Management Service determines a corrective course of action after it has been notified by the Measurement and Reporting component. Before taking the course of action, the Business Entity verifies that it is a valid (legal) action to take. Note that actions are simple ones such as creating trouble tickets. The Business Entity stores confidential business rules of the service provider, which enable it to verify that SLA parameters meet business goals/rules. Note that at this stage, the Business Entity has not been implemented and is performed by a human.

5. SLA Termination, which is responsible for terminating the SLA contract either automatically, based on certain conditions, or if the service customer or provider decide to terminate.

The SLA document provides for basic SLA aspects such as signatory parties, as well as aspects such as Measurement Directives, Functions, and Supporting Parties, which provide information on who can validate SLAs, and how.

This work provides a fundamental framework for SLAs in WS systems. Whilst monitoring is explicitly considered in these SLAs, the impacts of monitoring from the WS
Furthermore, whilst there is the ability to define who monitors SLAs and how, there is no mechanism to select the most efficient monitor from an available set.

### 3.3.6 CREMONA

Ludwig et al., 2004 presents an architecture (Cremona) for dynamic creation and monitoring of WS-Agreement SLAs for Web Services. Cremona is a middleware containing a Java library that provides implementations for the required WS-Agreement interfaces, management for agreements, and service environment abstractions to simplify implementation. The architecture is aimed at customised QoS guarantees, based on agreements that include consumer input. The aim is to have SLAs that define the QoS guarantee for particular time periods and usage patterns. The authors propose to extend the SOA publish-find-bind model to include these agreements, as shown in Figure 3.3.

**Figure 3.3: Cremona Architecture [34]**

When the service provider receives a request, it determines if it can meet that request (this may be some time in the future if the request is scheduled later). If it can, then the agreement is made. The system uses agreement templates in order to simplify agreement management. A monitoring agent in the system detects service agreement violations and predicts future violations based on these. Service reallocation may occur in these circumstances.

The Cremona architecture is composed of the following:
• An Agreement Protocol Role Management module, containing agreement provider functions for creating and storing agreements, agreement templates, and agreement consumer functions for implementing templates, agreement initialisation functions, proxies to the service provider’s agreement templates and agreements, and a function for monitoring;

• An Agreement Service Role Management module, containing agreement provider functions for agreement template creation and implementation plans, logic for determining whether to accept new agreements based on current loads, and monitors for ensuring current agreements are met. Provisions for the agreement consumer include functions for mapping agreements to client requests, and logic for requesting or accepting agreements based on current loads and compliance monitors; and

• A Strategic Agreement Management module, which provides higher level functionality including business rules on preferred partners and service types provided.

This work primarily differs from Keller and Ludwig, 2002 in that the service provider will accept and guarantee SLAs over time, based on current loads and incoming requests. However, since the system is designed to run below the business level, it only supports technical agreements such as throughput. The proposed approach does not consider the performance impact of monitoring. Furthermore, the actual implementation of many aspects, including monitoring, has not been performed. As such, there is no quantified performance impact.

3.3.7 Scalable SLA Monitors

The authors identify the need to monitor SLAs from a wide range of clients without consuming too many provider resources, and without causing violations to those SLAs, with special consideration for the scalability of SLA monitoring [13]. This scalability is in terms of number of clients and complexity of SLAs offered. The authors argue that as more, disparate SLAs are offered to potentially thousands of consumers, ‘naïve’ monitoring approaches will cause unnecessary resource consumption and possible SLA violation. To counter these concerns, the authors propose a generic method for developing scalable, adaptable SLA monitors.
Whilst the authors propose a generic overall approach of dynamic run-time adaptation of SLA monitors, they offer one specific approach, which is decreasing the frequency of monitoring SLAs that are not likely to be breached. This approach considers multiple clients with separate SLAs, and assigns more monitoring resources to those clients’ SLAs that are close to being breached. Whilst achieving a level of trade-off as well as pro-active management of monitors, this model assumes that monitoring costs are based on resource usage only, and so does not account for impacts from blocking interceptor type monitors, or monitors that share resources with the services they observe. In these cases, increasing the frequency of monitoring will have the opposite to the desired effect, reducing the QoS delivered.

The authors have identified the need to balance monitoring with QoS at run-time, in order to minimize SLA violations whilst maximising monitoring coverage. However, they have presented only one technique to achieve this, described above. Since the approach is generic, no implementation or evaluation has been performed. The authors have identified an implementation of their technique as well as an extension to their generic model as future work.

### 3.3.8 Privacy Agreement Monitoring

Benbernou et al., 2007 presents a method for monitoring the compliance of privacy requirements on Web Services by comparing run-time events to a pre-defined state machine of privacy requirements. They propose a framework for handling requirements on usage of private information (as opposed to requirements on access to private information).

The framework requires translation of privacy requirements into Linear Temporal Logic formulas as Private Data Use Flows (PDUFs), after which a set of probable privacy requirement breaches is defined, which may be extended later.

Privacy data requirements are specified as Rights providing authorisations for specific usage, and Obligations defining how data must be handled after being used. These privacy requirements are then used to create Privacy Agreements.

Privacy Agreements define policy-level and negotiation-level agreement parts. The Policy-level subsection contains a set of clauses making up a contract between the consumer and provider, as well as the definition of the elements required by the privacy data model.
and guarantees relating to this model. The Negotiation-level subsection contains the set of possible service behaviours with respect to privacy requirements, terms for negotiation (e.g. the set of possible actions for recourse after a privacy requirement is not met), the validity period of the agreement and penalties for not fulfilling the agreement.

The framework monitors the following system aspects:

- Private data, consisting of the core information (the actual private data in the system);

- Operations, which check if a service is performing the correct, expected action at any one time, or whether it is performing some other, unexpected activity;

- Roles, which define who should have access to the private data and who has received and will receive the data; and

- Temporal aspects, which monitor the flow of private data and compare it to stated requirements in the forms of either Right Triggering Time (data flow after an event and before another event), Right End Time (data flow ends at this time), Obligation Triggering Time (an obligation has been fulfilled at this time), or Obligation End Time (an obligation ends at this time).

The state machines modelling the privacy requirements define a state for each service in the system, and allowable transitions between states that show which data can move between which states, at a given time. An ‘agreement failure’ state is created which is lead to by any pre-specified non-compliant transitions.

Performance aspects of this monitoring technique are not discussed by the authors.

3.3.9 Modelling and Monitoring Web Service Requirements

Robinson, 2003 presents a method for monitoring Web Services to ensure they meet requirements. The authors argue that it is not possible to verify compliance to your own requirements when they are part of a multi-stakeholder requirement system, in which there may be conflicting requirements on the service provider. Therefore, the authors aim to verify run-time compliance to requirements, so that instead of ‘will the system meet my requirements?’ the question is ‘is the system currently meeting my requirements?’.
The authors divide service failures that will breach requirements into Silent Service Failure, Slow/Late Service, Incorrect Results, Service Spamming (a service that repeatedly calls other services for menial tasks e.g. posting catalogues), Faulty Interaction between services, Service Spoofing, and Denial of Service attacks.

The authors argue that consumers of service compositions require requirement-level feedback instead of transaction status or service availability reports, information on non-functional requirements, monitors that are able to be replaced or modified for each composition invocation, and formal verification of service compositions for business-critical tasks.

An Obstacle is defined as ‘an environmental condition that prevents requirements satisfaction’, and the authors claim that monitoring for obstacles is simpler than directly monitoring services for meeting requirements. To achieve their goals, the authors use the following approach:

1. Specify design-time goals and requirements (requirements are expressed in KAOS)
2. Specify design-time obstacles (using obstacle analysis or abuse cases) and monitors for each obstacle
3. Define the run-time system
4. Create design-time to run-time mapping
5. Define run-time monitors

Run-time monitors record all results, and the results are composed by top-level global monitors. Users can subscribe to the monitors to get results in SMTP, HTTP, and Rational Rose formats. Users can also query the monitors for specific information, although the communication details for the queries on monitors are not reported.

The performance impact of this system seems large due to the amount of monitors required, and the fact that the monitors have to respond to user requests as well as send information to subscribed users. However, performance is not considered in this paper. Furthermore, the system simply monitors everything, all the time - it cannot dynamically change its monitoring behaviour based on the results on monitoring.
3.3.10 FSA-based Monitoring

Li et al., 2006 presents a framework for monitoring and validating the run time interaction behaviour of services. The framework models service interaction constraints declaratively with FSAs, and verifies the interactions by intercepting messages between services.

The patterns to restrict interactions are based on Absence (invocations to the operation cannot occur), Existence (invocations to the operation must occur), and Bounded Existence (the number of invocations to the operation must be within a set range).

Sequencing is defined by patterns for Precedes (must occur before), and Leads To (must eventually be followed by).

Within the interaction and sequencing patterns, scopes can be set as any of Global (entire history), Before (history up to first occurrence of a message), After (after the first occurrence of a reply), Between-And (between consecutive occurrences of two messages in reply-call order), and After-Until (after a reply with no constraint on the response call).

OWL-S is extended with an Interaction Property Pattern (IPP) ontology, which is used to specify the interaction constraints.

Messages between Web Services are intercepted and analysed for adherence to the interaction constraints by the Monitoring and Validation Framework. The framework is divided into:

- A Message Monitor, which observes and analyses the SOAP message traffic in order to generate Interaction Constraint events;
- A Constraint Specification Manager, which uses an internal representation of the IPP based Interaction Constraints in order to determine and return all Interaction Constraints concerning a specified interaction event; and
- A Service Level Validation Manager, which manages the validation of all Interaction Constraints for a service. The Service Level Validation Manager uses Constraint Level Validation Managers to validate and report on each constraint. Constraints are obtained from the Constraint Specification Manager.

Rather than being a generic monitoring framework, this work is focused on detecting high-level violations in interactions with a service. Furthermore, the authors do not consider the performance impact of monitoring processes in order to detect these violations.
3.3.11 Timed Automata-based Monitoring

Raimondi et al., 2007 presents a method for monitoring Web Services for non-functional specifications, based on timed automata. This is achieved by mapping a set of specifications to timed automata, and using them to verify time stamped SOAP messages. The authors identify Latency, Reliability, and Throughput as recurring ‘specification patterns’, for translation into timed automata. The automata exactly characterise each of the specification pattern’s violations. The automata are verified against the sequence of communication events between the service consumer and service provider. The authors have implemented the system in Tomcat/Axis. UPPAAL is used to define the timed automata in XML. Acceptance checking is performed by a Java application, which outputs violations to a file. The complexity of the monitoring is $O(n^2)$ for automata size $n$. This work only handles the specification patterns of Latency, Reliability and Throughput, rather than being able to generically handle any possible quality type. A small prototype was implemented and the performance overheads were reported as being negligible, however the authors did not quantify these overheads, and performance impacts of a more complex system were not reported.

3.3.12 Consumer-level Web Service Monitoring

The authors identify the need to test Web Services for functional correctness at bind-time and run-time [15]. This testing is from a service consumer’s perspective. Web Services are classified according to how they may be tested: with test oracles, with lookup tables, by probing or by eavesdropping. This classification is based on whether services are stateful or not, and the complexity of the relationship between input and output.

This is a generic methodology to be applied when designing testing systems for Web Services, from the consumer’s perspective. There is no consideration of the performance impact or other cost of performing testing.

3.3.13 SALMon

The authors have identified the need to monitor services and service compositions for violations of SLAs [4]. The authors also identify the need to be able to modify SOA-based systems at run-time in order to ensure QoS constraints are met. To meet these needs,
the authors propose the SALMon tool, aimed to ensure SLA requirements are met by monitoring and adapting monitored systems at run-time.

The authors adopt an extension of the ISO/IEC 9126\(^1\) quality standard, giving Web Services both technical and non-technical quality characteristics. Technical characteristics are Functionality, Reliability, Usability, Efficiency, Maintainability and Portability. Each of these technical characteristics is defined by a set of sub-characteristics. For example, Functionality is composed of Accuracy, Suitability, Interoperability, Security, Auditability and Functionality Compliance. Non-technical characteristics are divided into those about the service, such as Pricing and Testability, and those about the provider, such as Reputation and Security Policies. Each of these sub-characteristics is linked to a set of measurable attributes. For example, the Accuracy sub-characteristic of Functionality may be linked to a test oracle for validation.

However, the authors focus only on those quality characteristics from ISO/IEC 9126 that they consider ‘monitorable’, such as performance and availability.

Monitoring is achieved with the SALMon architecture. This architecture is composed of Monitors, Analysers and Decision makers, as shown in Figure 3.4. The components in the architecture are each implemented as Web Services. A central database stores SLA-sourced requirements and monitoring results.

\(^{1}\)http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749
A partial prototype has been developed, which excludes the Analyser component, and includes only one Monitor, used to measure response time. However, there is no verification or validation reported based on this prototype, and no performance testing reported.

Whilst the authors have identified the need to perform monitoring on a Web Services system with a minimal performance impact, this work does not measure the performance impact of their approach. Furthermore, this work does not consider balancing monitoring with delivered QoS, and only considers complete monitoring coverage as a suitable monitoring solution.

### 3.3.14 WSFL

The authors note the need to automate SLA management in WS-based systems [58]. This involves automation of SLA contract binding, SLA monitoring, and compliance control. The authors argue that for SLAs to be monitored effectively, the monitoring must be distributed across provider’s and consumer’s sites. This would allow for a more precise measurement of the consumer’s delivered QoS, as well as measurements of QoS delivered between services in a composition.

To achieve this automated SLA management, the authors propose a standard SLA
specification language, a monitoring technique, and a monitoring management agent.

The SLA specification language is intended to be generic, such that it can be adopted for any SLAs, and precise, such that automated management can be performed based on the SLAs with no concerns over ambiguity. Each SLA consists of a contract time range and a set of SLOs. In turn, each SLO consists of a set of clauses, which are based on monitoring results. Monitoring results are specified with a location (e.g. “server hosting service A”), an evaluation trigger (e.g. “when service A is invoked”), an aggregation function (e.g. “mean response time of last five transactions”), an evaluation function (e.g. “mean response time must be less than 5 seconds”), and a trigger action to perform after evaluation (e.g. “refund user when average response time over 5 seconds”). Complete details of the SLA specification are provided in the author’s previous work [59].

The monitoring technique uses SOAP interceptors to record information on service invocations and responses. These SOAP interceptors are implemented inside each SOAP server, as software modifications of those servers. It is assumed that monitoring compositions can be achieved by reading log files from process (e.g. Web Services Business Process Execution Language (WS-BPEL)) servers.

The monitoring management agent is intended to automate the creation and observation of SLAs. This is achieved by creating management process flows in the Web Service Flow Language (WSFL). These WSFL management process flows describe the SLA-sourced management actions required, and are executed by appropriate management agents.

This work proposes an automated WS monitoring solution for the purpose of monitoring and managing SLAs. However, the monitoring implementation is fixed (as modifications to SOAP servers). This reduces the possible set of monitoring techniques available (such as probes). Furthermore, the performance impacts of monitoring have not been discussed. This is especially important when monitoring is being performed at both the provider and consumer side, and the results of monitoring have a direct impact on compliance with SLAs.
3.3.15 SLA Monitorability

Skene et al., 2007 presents an evaluation of the monitorability of SLAs in application service provision scenarios. The authors argue that if the targets of an SLA are not able to be monitored effectively, then the SLA is not likely to be able to be enforced.

The authors divide SLAs into being either unmonitorable, monitorable by one party, monitorable by both parties (provider and consumer), or monitorable by a third party. Trusted third parties are defined as those that have no financial interest in the event and are able to directly monitor SLA targets.

The authors claim that the most monitorable Web Services solution is one where both the service consumer and service provider monitor all interactions. Since the exact value of the targets being measured will never be recorded by the monitors, the authors have introduced a statistical process for determining the likely error of each monitor so that either party of the SLA can verify their own log against the other party’s log and see if the other party’s log is within the correct range, including measurement error (out of this range would imply malfunctioning monitors or falsified logs).

Whilst this work considers how Web Services may be monitored in order to ensure trust, there is no actual monitoring solution provided by the authors. Furthermore, the only quality types considered are time-related, such as throughput and turnaround time.

3.4 Monitoring Service Compositions

This section reviews those research efforts that present WS monitoring solutions that are targeted at WS compositions or processes. These may be generic processes, WS-BPEL processes, or other specifically defined WS processes. We have also extended the definition to grid-based applications and processes. Various constraints are monitored by these systems, including latency (e.g. a response from service X must arrive within 30 seconds after a request), throughput (service X handles at least 50 requests per minute), and behaviour (Shop service sends a verify credit card message to Bank service immediately after the Shop service receives a Purchase request from a consumer).

WS monitoring may be performed with methods such as intercepting proxies, test probes, or eavesdroppers. WS composition monitoring is either performed by SOAP mes-

Mahbub and Spanoudakis, 2004, Baresi and Guinea, 2005, Barbon et al., 2006, Lazovik et al., 2004, and Gan et al., 2007 each present methods for annotating or mapping WS-BPEL descriptions to constraints that are managed by a monitoring system. Baresi and Guinea, 2005 presents a method for dynamic monitoring of WS-BPEL processes by adding QoS related monitoring rules to those processes. Monitoring rules are defined in WS-CoL (WS-Constraint Language) and imposed on WS-BPEL processes. A pre-processor weaves the WS-CoL rules into the WS-BPEL description at deployment time. A monitoring agent is then used to verify the WS-BPEL process via either pre- or post-condition evaluations. Baresi and Guinea, 2011 extends this by making WS-BPEL processes self-monitoring. This work uses both local and remote (third party) monitors to observe the interactions within WS-BPEL processes, and gives priority levels to monitoring, in a similar way to Baresi and Guinea, 2005. Similarly, Lazovik et al., 2004 presents a framework for linking business rules (assertions) defined in XSRL with WS-BPEL processes. The framework models business processes as a graph of States (current state of process execution) and Actions (business activities that are the transitions between States). Actions are executed on behalf of Roles (Roles are generic businesses, e.g. Travel Agency). Assertions may be made over individual aspects of a composition (e.g. Hotel Price < $300p/n), or an entire composition (e.g. Total cost < $10,000). Assertions may be defined at the business process, role, or provider level. Barbon et al., 2006 presents a WS-BPEL process monitoring solution that allows the process designer to specify constraints in a custom language, which is then automatically translated into Java and used to monitor and administer both WS-BPEL process instances and the lifetime of every instance of each WS-BPEL process. Gan et al., 2007 presents a framework for designing Web Services processes as Unified Modelling Language (UML) sequence diagrams, and translating those sequence diagrams into Non-deterministic Finite Automata (NFAs).
These are used by a monitor to verify Web Services processes at run-time. Mahbub and Spanoudakis, 2004 presents an event calculus based framework for monitoring Web Services requirements. The framework is used to automatically translate a WS-BPEL specification into a set of event calculus specifications, which are used to validate service processes at run-time.

Rouached and Godart, 2006 presents a method for analysing execution logs in order to validate the behaviour WS compositions, using discrete event calculus. The verification is performed by describing the expected properties and the execution log in discrete event calculus, and verifying the latter against the former using SPIKE (an automated induction-based theorem prover).

Ezenwoye and Sadjadi, 2006, Zeng et al., 2007 and Singh et al., 2006 present methods to detect and replace faulty services by monitoring service processes, with particular emphasis on the performance of their monitoring solutions. Ezenwoye and Sadjadi, 2006 presents a technique for monitoring WS-BPEL processes in order to detect and replace faulty or disconnected component services at run-time. Zeng et al., 2007 presents a method for high-performance monitoring of services and processes in real-time. To achieve this, requirements are modelled as a set of Event, Condition, Action (ECA) rules, which are executed with the system, and used to validate monitoring results. Singh et al., 2006 presents a method for flexible monitoring of a distributed system. This technique models a business process in a data-flow graph, which is used to detect process violations.

Truong et al., 2006 presents a framework for monitoring and analysing the QoS of grid-based services, in order to detect faults of and optimally allocate resources to services.

Narendra et al., 2008, Panahi et al., 2008, Benharref et al., 2009, and Zhang et al., 2007 each present a method to minimise a particular cost of monitoring WSs. The cost to be minimised varies from business-level utility to low-level computer resources. Narendra et al., 2008 presents a method for determining ideal locations for a set of finite monitors in a business process, by balancing the cost and benefits of monitoring. Panahi et al., 2008 presents a framework for monitoring and managing service processes in order to increase their reliability and performance. This is achieved by calculating optimal monitoring locations, and using monitoring results to diagnose faults and reconfigure faulty processes.
Benharref et al., 2009 present a method for identifying WS faults through eavesdropper-type monitors and log analysis. The depth of monitoring is varied in order to identify the cause of faults either as an entire service process, or components in a process. The authors have attempted to minimise bandwidth usage of monitors. Zhang et al., 2007 present a technique for composing a service process such that the QoS of that process and the cost of the monitoring it are optimised.

Qiang et al., 2006 presents a mechanism for **detecting and handling process exceptions** in SOA-based systems. Exceptions are classified as events that disrupt normal message flows between Web Services. The authors propose a standard definition for exceptions, as well as standard methods for handling and raising exceptions between Web Services.

Bai et al., 2009, da Cruz et al., 2004, Leitner et al., 2010, and Molina-Jimenez et al., 2004 present methods to monitor Web Services systems to validate **Service Level Agreements**. Bai et al., 2009 presents a technique for generating monitors to monitor services as a WS-based system evolves at run-time. These monitors observe services to ensure they match required specifications from SLAs. da Cruz et al., 2004 presents a Web Services utilisation monitoring solution based on interception and logging of SOAP messages by SOAP intermediaries. The architecture, WSLogA, is based on analysis of the utilisation logs. WSLogA uses both Active (able to modify the SOAP message) and Passive (cannot modify the SOAP message) SOAP intermediaries. Leitner et al., 2010 presents a technique for monitoring SLAs with the goal of reducing the probability of future SLA violations, by monitoring WS compositions in order to predict and eliminate future SLA violations. This is achieved by performing regression analysis on historical QoS records, based on the current QoS being delivered by a composition. Molina-Jimenez et al., 2004 considers SLA measurement from the consumer’s perspective, and presents a framework for creating and monitoring SLAs, which provides for the ability to state how SLOs will be monitored, and by whom. The framework considers the QoS that will be delivered from each consumer’s perspective, based on that consumer’s location.
3.4.1 Control Point Selection for Monitoring

The authors identify the need to continuously monitor run-time service communication in order to verify compliance to regulations such as privacy and security acts [39]. The authors propose a method to insert monitoring points in a business process in order to meet this need. This technique involves a tradeoff between monitoring cost (including human resource costs) and business value.

The authors identify the need to trade off between the cost of evaluation (in monetary terms), the potential for non-compliance, and the number of monitors. The authors focus on higher-level business requirements such as privacy and security. In these cases, requirements for monitoring are taken from various sources such as laws and regulations. Each monitoring requirement is assigned some weight that represents its importance.

The authors optimise either to minimise the number of control points, or to balance the cost of monitoring with the cost of non-compliance from not monitoring.

Optimising according to the number of control points attempts to minimise the number of monitors used to achieve required monitoring coverage. This optimisation takes into account that some monitors may be human, and therefore costly. This optimisation is a direct trade-off between the cost of monitoring in terms of wages, infrastructure, etc., and the cost of not monitoring in terms of non-compliance to laws and regulations, all in monetary terms. The optimisation problem is solved using an approximation algorithm that calculates the net value of monitoring everything, and then starts removing monitors for the lowest value requirements, calculating the net value, until the net value starts to decrease.

Optimising according to the net cost of monitoring and non-compliance attempts to find the optimal set of monitoring locations (which are analogous to monitoring activities or actions) given a set of available monitoring resources. For example, if there are three monitors available to a provider, what requirements should they be set to monitor? Note that there is an assumption here that monitors are either human or fixed resources rather than simple software components that may be duplicated. The optimisation is solved using an approximation algorithm that selects only monitoring activities with a lower cost than the cost of non-compliance, and then monitoring activities are selected in order of decreasing value until there are no monitoring resources remaining.
Whilst this work considers the cost of both monitoring and not monitoring for optimisation, it does not meet all of our requirements for a monitoring system. Most importantly, the work is designed primarily for human resources as monitors. This is fundamentally different in that the monitoring resource numbers are fixed, expensive, and are assumed to be able to monitor any activity. At the lower level, the cost of monitors is provided in dollar terms rather than cost derived from the impact on QoS. Monitors are binary (on or off) only, rather than having a set of possible monitoring levels, and the monitoring cost is fixed, with no overhead costs or indexing to monitoring load.

### 3.4.2 Optimal Web Service and Monitor Selection

The authors identify the need to monitor Web Services in order to measure and manage their run-time performance, whilst minimising the cost (in terms of QoS impacts) of monitoring on the system [68]. To meet these requirements, the authors propose optimal selection of monitors and services in a whole composition such that the total QoS of monitors and services is best met.

The set of system QoS constraints is used to generate a utility function that describes the utility of meeting each QoS level in the system. This utility function is used when selecting concrete services in a composition, in order to obtain an optimal QoS profile. When determining the best set of concrete services, the cost of monitoring is taken into account. For example, there may be two fast services on different servers that require two monitors, or two slower services on one server that can share a monitor, and therefore have a lower total cost.

The problem of selecting an optimal set of services and monitors is modelled as a Weighted Set Covering (WSC) problem, and solved using a greedy algorithm. This algorithm calculates the cost efficiency of monitors, as a ratio of monitoring coverage to monitor cost. Monitored compositions are then designed starting with the most efficient monitors.

The authors have prototyped their technique, and measured the cost savings for their service and monitor selection algorithm against the case when services are selected while the cost of monitors is not considered. A reduction in QoS impacts of 60-70% is achieved, although the authors note that this is dependent on the target system and monitoring
The authors of this work identify the need to minimise the cost of monitoring in a WS system. However, this work takes a static, one-dimensional, user-defined cost for each monitor when optimising. This does not allow for each QoS impact of a monitor to be considered, and does not account for the impacts of monitors changing with the activities those monitors perform. For example, the assumption in this model is that the cost of using a monitor to observe one service is the same as the cost of using the same monitor to observe two services. Furthermore, the authors search for complete monitoring coverage, rather than considering systems in which it may not be beneficial to monitor some qualities of service.

### 3.4.3 Dynamic BPEL Monitoring


The authors list the set of unsolved problems in WS-BPEL dynamic monitoring as allowing the application designer to:

- Select the QoS values of interest for monitoring;
- Select how those QoS values are measured and gathered;
- Change service monitors at run-time; and
- Change the level of monitoring at run-time in order to balance monitoring with performance.

The approach uses Monitoring Rules, external to WS-BPEL, which are used to control each WS-BPEL process. Each Monitoring Rule can be enforced on different WS-BPEL processes, allowing reuse of the rules. The Monitoring Rules are defined in WS-CoL (Web Services Constraint Language), a Java Modelling Language extension.

A Monitoring Manager intercepts all WS-BPEL message flows and executes any pre-specified rules that are required at each step. If the Monitoring Manager detects a fault, it notifies the WS-BPEL process. Recovery from faults is not handled, however.
The Monitoring Rules are weaved into the WS-BPEL specification at deployment time by a BPEL² pre-processor, resulting in a BPEL² specification, which is then interpreted by a Monitoring Manager. Rules are weaved into the WS-BPEL specification as follows:

1. An initial configuration call to the Monitoring Manager is made with all monitoring rules and information

2. Post conditions are replaced with a call to the Monitoring Manager, which verifies the condition before executing the next WS-BPEL service call

3. Pre conditions are replaced with a call to the Monitoring Manager, which verifies the condition before executing the WS-BPEL service call

4. Scope invariant rules are translated to post conditions

5. A final Release call to the Monitoring Manager is made

The rules are created with an equivalent of debug levels (1-5), which allows for performance versus monitoring trade-offs at run-time. We consider this a fairly high level of abstraction, since the monitoring level must be set for a whole service, instead of setting for individual QoS types, or a particular quality aspect to monitor across all services.

This work is unusual in that it considers and adapts to the performance impact of monitoring. However, whilst the method allows for performance versus monitoring trade-off, the implementation is a basic five levels of monitoring for services. This would be more beneficial if monitor levels were set per QoS. That is, rather than assigning an importance value between one and five for all actions that a monitor takes, directly assigning a monitoring level to every action a monitor takes results in a finer level of control. Furthermore, this work only allows for a designer to select the monitoring level, rather than having monitoring levels automatically set based on current or recently delivered QoS levels, and there is no automatic feedback mechanism from delivered QoS to indicate when changes in monitoring configurations should take place. Finally, the work does not cover WS systems using dynamic binding, since service calls are hard coded into the BPEL² file.
3.4.4 Self-monitoring BPEL

Baresi and Guinea, 2011 identifies the need to monitor WS-BPEL processes that make use of some external (third party) components, in order to detect functional faults or when the delivered QoS becomes unacceptable, and re-compose the WS-BPEL process to remove the offending component. To achieve this, the authors propose a *self-supervising WS-BPEL process*. This process uses ‘Supervision Rules’, which define which process components to monitor and how, and what actions to take when monitoring detects a QoS or functional violation.

The authors’ Web Service Constraint Language (WSCoL) [8] is used to define pre- and post-conditions for component services. ECA rules are used to trigger recovery actions when constraints are breached at run-time, and those actions (such as replacing faulty service) are defined using the authors’ Web Service Recovery Language (WSReL).

Monitoring can be enabled or disabled by setting monitoring priorities for each component service. For example, each service may be given a priority between 1 and 10, and at some point at run-time, the monitoring priority level may be set to 5, meaning that all services with a priority level greater than 5 will be monitored. Time period thresholds are also provided for enabling and disabling monitoring. Finally, a white-list of services can be set that never require monitoring due to being hosted by trusted providers.

Recovery is divided into *local* and *backward* recovery. Local recovery involves repairing a composition mid-execution, and continuing from the point of repair, whilst backward recovery involves rolling back system state to before the fault occurred, and then repairing the composition and proceeding forward again. Full details of these techniques are not relevant to monitoring, and so are omitted here.

The authors have measured the performance impact of their approach for a prototype system, in terms of response time. Depending on the type of monitoring used, the response time increase varied from approximately 10% to 100%.

This work considers the performance impact of monitoring and provides a mechanism to reduce that impact by setting a priority for monitoring each service. However, as with the authors’ previous work [8], there is no mechanism to automatically adjust those monitoring priorities, and no mechanism to balance monitoring with QoS.
3.4.5 LLAMA

Panahi et al., 2008 presents a framework for monitoring and managing SOA business processes. Their intelLigent Accountability Middleware Architecture (LLAMA) framework provides a system for the initial setup and configuration of monitoring and management agents, diagnosis of failures and faults, and automatic re-configuration to recover from these failures and faults. The authors have identified the need to ensure monitoring is not overly invasive.

The goal for this work is to increase reliability and performance of WS-BPEL business processes, and ensure that service providers provide accountability for their delivered QoS. This is achieved with the LLAMA framework, which provides mechanisms for:

- Determining optimal monitoring locations;
- Diagnosing faults in business processes; and
- Repairing faulty processes via reconfiguration.

The framework monitors correctness and performance of processes. This is achieved using existing monitors. Monitoring data is aggregated and analysed in order to perform fault diagnosis. Monitoring is performed on atomic services and service compositions, and resources such as networks and host hardware.

Monitoring configuration is performed by selecting those monitoring agents in the system that can perform required monitoring actions with the least cost, and includes consideration of monitors with overlapping capabilities. Details of the algorithm for selection are not reported, however the authors suggest the use of basic algorithms such as greedy set covering techniques.

Fault diagnosis is performed by using Bayesian analysis to identify the root cause of measured failures. In this case, root cause may be a faulty service, a faulty service relationship, or a lack of system resources. For example, if a server hosting service $a$ goes from 30% to a 99% CPU usage, and the response time of a service process including that service suddenly increases, then the fault diagnosis engine may identify the need to re-configure the process in order to replace service $a$.

System or process reconfiguration occurs after a fault or failure has been detected in the system. For the example above, the reconfiguration engine may replace service $a$ with
an equivalent service $a'$ on a different server, and monitor to ensure that the response
time of the process has decreased to an acceptable level. When replacing services, the
reconfiguration engine uses services that do not share the resource identified to be faulty.
For example, if a network is diagnosed to be the root cause of a failure, then a service on
that network will not be replaced with one on the same network.

The LLAMA framework has been implemented as an extension to the Mule Enterprise
Service Bus\(^2\), and its accuracy and performance impact have been measured with a small
case study system. Monitors could either be set to push data to monitoring agents as it is
generated, or wait for agents to request summaries (at lower time intervals). The authors
measured a 1-7\% impact on response time for the push method, and ‘negligible’ impact
for the pull method.

Whilst identifying the need for an efficient monitoring solution, this work does not
allow for a complex monitoring versus performance trade-off. Rather, a fixed monitoring
coverage requirement is met by assigning the most efficient set of monitors. The authors
have measured the impact of these monitors for a small case study, however details of the
monitoring implementation are not provided.

### 3.4.6 FSM-based Web Service Behaviour Monitoring

The authors identify the need for monitoring Web Services and service processes to identify
WS faults [12]. To achieve this, the authors investigate the use of passive testing for WS
fault detection. This passive testing is classified as both ‘online’, where tests are performed
as services are executed, and ‘offline’, where logs are analysed some time after execution.

To achieve their goals, the authors propose an extension to the service-oriented ar-
chitecture, with ‘Observers’ for gathering traces through monitoring. Observers report
to both service providers and service consumers, and are implemented as Web Ser-
vices. Management agents may use the monitoring data for fault diagnosis, and isolation
or correction as available.

The authors’ goal is for monitoring to not interrupt the messages of monitored services,
and for monitoring to have minimal impact in terms of bandwidth consumption.

The framework provides for single monitors to observe atomic services, and sets of

\(^2\)http://mule.codehaus.org/display/MULE/Home
monitors for observing service compositions. Monitors may be white-box ‘local’ observers, or black-box ‘global’ observers. The choice of which to use will depend on the monitoring depth required. For example, if a service composition is hosted by one provider, then a black-box monitor may identify a fault within that composition, whilst local monitors may be required to diagnose that fault. Monitors are generated based on WS-BPEL and WSDL description files.

Services and compositions are (manually) modelled as finite state machines, or finite state machines with time constraints, where appropriate. These finite state machines are used by monitors to determine whether services and compositions have behaved correctly.

The authors have implemented both a simple scenario with a single monitor and service and no timing constraints, and a more complex scenario with a service composition, a set of mobile monitors, and timing constraints. Each scenario was measured for its validity and performance impacts. For the single monitor scenario, monitoring doubled the network traffic used. This is due to the 1-to-1 ratio of monitored executions and monitoring reports, and roughly equivalent message sizes. The resource usage of the monitor was observed in terms of CPU and RAM utilisation, and was considered minimal (less than 5%). For the more complex scenario, network load increased with the number and location of monitors. Mobile agents collocated with services minimised total impact. This is due to the reduction in bandwidth consumed, and relies on the minimal CPU and RAM utilisation of monitors and analysers.

Whilst the authors have considered the performance impacts of monitoring using their approach, these impacts are measured in terms of resource consumption only, rather than impacts on delivered QoS from services. As such, there is no balancing between monitoring coverage and delivered QoS.

### 3.4.7 Web Service Composition Monitor

The authors identify the need for service providers to minimise the occurrence of SLA violations. Rather than analyse monitoring records after SLAs have been violated, the authors propose a method for predicting when SLAs are going to be breached, and re-composing a service in order to avoid this [31].

The authors use a framework that consists of a Composition Monitor for observing
service compositions, which reports to an SLO Predictor for predicting when SLAs will be breached. When the SLOs Predictor predicts that an SLA will be breached, a Composition Adaptor is triggered to re-compose the affected service composition.

Monitors are generated for events in a composition. For example, a monitor may be generated that calculates the response time of a composition by observing the time between the initial service request and final service response. The method for implementing and placing these monitors is not reported.

SLA violation prediction is performed using regression analysis on historical and run-time performance data. At each point in a composition, the current run-time performance is validated using a Checkpoint Database. If, based on records of previous executions using the planned composition, it appears unlikely that some SLA will be met, re-composition is triggered.

Re-composition may be changing a parameter for a service execution (e.g. to request higher priority processing or faster shipping), or re-binding some services in the current composition. Composition can only replace component services with other equivalent component services, rather than re-composing an entire service with a new structure. The ability to re-compose in this manner requires knowledge about the performance of each component service and its parameters.

The authors have partially implemented their framework and verified it using a case study system. Whilst demonstrating the technical correctness of the approach, the authors did not measure the performance overhead of any part of the system.

3.4.8 Automated SLA Monitoring Deployment

The authors propose an abstract framework for creating and monitoring SLAs between service providers and consumers [38]. This framework allows for an SLA to state how and by whom a QoS will be monitored. The authors divide quality aspects into computational (e.g. time for a service to compute a result) and communication (e.g. time for a result to be transmitted) types. The communication path through the cloud (the string of ISPs between the service provider and service consumers) is explicitly considered as part of the communication resources. The authors consider only non-functional aspects in SLAs.

The proposed model calculates ‘points of presence’, which are those ISPs on which
service consumers will have a guaranteed QoS level on the communications infrastructure. These also allow a service provider to tailor an SLA to each service consumer, according to their network point of presence.

The authors classify monitors into active (interceptors and probes) and passive (e.g. eavesdropper) types. They consider the location, owner and administrator of monitors, as well as what information each monitor can provide. With these parameters, the authors define four general approaches to monitoring:

1. One monitor collocated with each service consumer
2. One probe per service consumer, sharing that consumer’s point of presence
3. One monitor collocated with the provider
4. A set of interceptor-based monitors between the provider and each point of presence

The proposed architecture involves the use of a third party to provide monitoring. This is to ensure trust without having to monitor at both the consumer and provider endpoints. Monitoring is divided into a set of measurement services which report to an evaluation service. The proposed solution relies solely on third-party probe-type monitors. Whilst mitigating potential trust issues, this limits the possibilities of monitoring. For example, it is still not possible to prove what QoS was delivered to a consumer. Furthermore, messages that breach some condition such as security or privacy cannot be intercepted en-route.

Whilst identifying that the rate of probing for monitoring should vary according to SLAs, the authors have not mentioned the possible performance impact of this approach, or how that impact may be minimised or traded off with monitoring frequency.

### 3.4.9 MDA-based Web Service Monitor Generation

The authors identify the problem of assuring the dependability of service compositions as they are composed and re-composed at run-time, with component services from possibly unknown (to the consumer), external providers with whom trust is not established. Therefore, the authors propose continuous online quality control, to ensure services stay both available and reliable [6].
Rather than manually developed monitors that must be changed with new system changes or monitoring requirements, the authors propose the use of a Model-Driven Architecture (MDA) for automatic generation of monitors. Both a Service Model (system description) and a Policy Model (SLAs) are used to generate a set of Sensors (monitors) for data gathering and evaluation. The work requires services to be described by WSDL and policies to be specified in WS-Policy in order for monitors and analysers to be generated.

The authors have identified the need for different types of monitors to cover different monitoring scenarios, such as interceptor based monitors to verify reliability, and probes to request resource usage rates. Monitors are defined by their system level (e.g. white-box or black-box, or whole-of-composition monitors) and their capabilities (what qualities can be measured, and how).

The authors have also identified that it is too costly to monitor everything in the system. As such, monitoring is configured in one of three ways. **Dependency-based** monitoring configuration involves identifying services that are co-dependent and monitoring their interactions. This requires an extension to WSDL that provides input and output dependencies between services. **Coverage-Based** monitoring configuration involves selecting monitors to achieve some desired coverage. Here, coverage is on either services or compositions, and measures which WSDL interface elements (e.g. message/operation) and WS-BPEL control constructs (e.g. sequence, iteration) have corresponding monitors. **Goal-based** monitoring configuration involves generating monitors that will observe the critical components in a system. For example, a response time monitor may be set to observe the longest composition or slowest service. However, only dependency and coverage based methods are covered in this paper.

Monitors are generated semi-automatically, where designers provide requirements for quality types to monitor (e.g. to observe functionality or performance) and coverage criteria (which monitor or process to observe, and where). This is used to determine which services should be monitored, and how. Monitors are then automatically generated and deployed.

Policies are composed of a Type (classification), Subject (target service) and set of Assertions (verification rules). There are three Types of policies: **Data**, defining rules on dependencies; **Functional**, defining rules on functional correctness; and **Performance**,...
defining rules on performance attributes such as response time. A policy enforcement engine is used to process these policies, and verify they are met using incoming monitoring data.

The system has been implemented and tested with a small prototype. Overhead analysis was performed, and the impact was measured to be below 5% for the prototype system. For the prototype system, all monitors were ‘process listeners’, (which appear to be eavesdroppers), and analysis was performed on a remote, dedicated machine.

The authors have identified the need to minimise the disruption from monitoring. This is achieved by monitoring only those system elements that are related to SLAs. However, their work does not consider the cost of monitoring impacts in terms of SLA violations. Furthermore, the work uses a rigid set of quality types that does not allow for more complex monitoring criteria, such as service-specific security requirements.

3.4.10 MI-Eh

Qiang et al., 2006 presents a mechanism for detecting and handling exceptions in service oriented systems. Exceptions are classified as events that disrupt normal message flows between Web Services. The authors claim that service oriented systems raise the following problems for monitoring:

1. lack of a standard way for raising exceptions and passing them through service chains
2. lack of a method to handle exceptions, particularly:
   (a) How to abstract away unnecessary information from the exception recipient;
   (b) How to ensure that important information is displayed; and
   (c) Preventing overwriting of old/current exceptions with new ones.

The authors propose solving these issues with the introduction of the MI-Eh framework. The framework provides systems for:

- Uniform exception definitions, including methods for specifying how far exceptions can be raised;
- Dynamically extending exception declarations; and
• Defining methods to customise the analysis of groups of exceptions, e.g. selecting only particular exception types, or tracing a single exception from its roots.

The authors propose a system for automatic propagation, logging and buffering of exceptions between Web Services (but do not provide or define a solution that meets this proposal).

Whilst the framework may allow for targeted monitoring of an SLA system in order to reduce unnecessary monitoring coverage, the authors have not mentioned the possible performance impacts of their monitoring framework. Furthermore, there is no consideration of disabling monitoring when it might cause exceptions.

3.4.11 Event Calculus-based Web Service Monitoring

Mahbub and Spanoudakis, 2004 presents an event calculus based framework for monitoring Web Services requirements. The framework is used to automatically translate a WS-BPEL specification into a set of event calculus specifications. The framework can only translate basic WS-BPEL activities (invoke, receive, and reply). At this stage the framework cannot translate WS-BPEL throw or wait activities. Deviations are divided into Inconsistency of Recorded Behaviour (a fault discovered in the log), Inconsistency of Expected Behaviour (a missing event), and Unjustified Behaviour (an event that should not have occurred). Detection of these deviation states is based on reasoning on the event calculus defined by the WS-BPEL requirements. A Monitor is used to detect any of these deviations in the running system by monitoring the log of events.

This method allows for violations in requirements on WS-BPEL specifications to be detected by analysing event logs. However, the method is not generic in that it does not allow for any non-WS-BPEL requirements to be modelled. This means requirements sourced from SLAs or corporate governance, or other software requirements cannot be monitored in the system. Furthermore, the system only detects violations in behavioural requirements of service compositions. This means other process requirements such as QoS are not monitored. The system does not diagnose requirements deviations that it discovers. Doing so would allow for faults to be quickly (and possibly automatically) rectified. Finally, the performance impact of monitoring is not measured or discussed.
Barbon et al., 2006 propose a technique for monitoring WS-BPEL processes. This paper differs from the other WS-BPEL process monitoring solutions in that the monitoring is performed both on WS-BPEL process instances, and on an abstract WS-BPEL process, so that the result of monitoring can be tailored to each stakeholder. For example, a consumer may want the results of a single or handful of WS-BPEL process instances, whereas a manager would want the cumulative results of monitoring all instances of that WS-BPEL process. Here, the abstract monitoring is simply an aggregation of the monitoring results of each instance of that WS-BPEL process.

Monitors are described using the Run-Time Monitor specification Language (RTML), a language created by the authors for this purpose. The RTML is then automatically translated into Java applications for execution. The RTML expresses boolean, statistical, and temporal properties on service processes.

Business logic defined in RTML is separated from the actual monitoring through the use of a Monitoring Framework that is separated from the WS-BPEL Execution Engine. The authors have demonstrated their technique using Active WS-BPEL. Figure 3.5 shows the relationship between the existing WS-BPEL Execution Engine (white background) and the author’s contributions (grey background).

![Figure 3.5: BPEL RTML Extension [7]](image-url)
The new components consist of:

- The Monitor Inventory, which stores all abstract or class-level monitors;
- The Monitor Instances, which store all instantiated monitors;
- The Runtime Monitor, which creates and terminates Monitors;
- The Mediator, which allows communication between the Runtime Monitor and the Queue and Process Managers via message interception and process creation/termination at the WS-BPEL engine side; and
- The Extended Admin Console, which extends the Active WS-BPEL Admin Console to provide the user with information on the existing Monitors.

The RTML contains constructs for defining events such as messages between services or the creation of process instances, and monitoring formulas describing rules to be validated with monitoring.

The performance impact of monitoring is not reported by the authors. Although the framework is based on interception and does not probe services, it sits on the same server as the WS-BPEL server, and creates its own WS-BPEL processes, which may impact performance. Furthermore, there is no consideration of balancing monitoring coverage with monitoring impacts.

### 3.4.13 Monitoring BPEL Processes for Robustness

Ezenwoye and Sadjadi, 2006 presents a framework and approach for monitoring WS-BPEL processes and replacing faulty Web Services at run-time. Timeout and fault events are monitored from within the WS-BPEL process, and monitors invoke a Web Services handler to replace any faulty services. The WS-BPEL definition for a monitored process is extended to include fault handlers, containing fields for catching faults and sending the service request to an alternate service, and event handlers, containing fields for timeouts and proxies containing alternate services. Note that although services are replaced at runtime, they are replaced from a list of equivalent services that is created at design-time. Service compositions are not replaced (e.g. replacing ‘verify and bill’ service with ‘verify service’ and ‘bill service’).
Whilst it would be possible to monitor some QoS (at the very least throughput and response times, etc.) in this system in order to change service providers when QoS drops below acceptable levels, performance impacts are not considered by the authors. Furthermore, there is no consideration of selecting monitors for anything other than complete monitoring coverage.

3.4.14 UML-based Web Service Conversation Monitoring

Gan et al., 2007 presents a method for monitoring Web Services conversations, and verifying them based on UML sequence diagrams which have been translated to non-deterministic finite automata. The authors’ goal is to create a framework that is non-intrusive, allows for dynamic WS discovery, and supports various message exchange types.

The UML sequence diagrams used by the authors are a subset of UML 2.0 sequence diagrams, and are used to express either Safety (e.g. ‘goods must not be shipped without payment’) or Liveness (e.g. ‘goods must be shipped after payment’) properties via an extension to the sequence diagram containing Negate and Assert operators.

Negate operators, which express Safety, are wrapped around sequence diagrams that should never occur. Assert operators, which express Liveness, are wrapped around sequence diagrams that must occur.

The authors have implemented the system using the WebSphere Process Server and Integration Developer. Figure 3.6 shows the architecture for the implementation.
Service consumers first create UML Sequence Diagrams for their intended Web Services applications using a GUI provided by the Property Manager. The Monitoring Manager then translates the Sequence Diagrams into automata. At run-time, the Message Manager takes messages from the SCA Message Handler, which are then passed onto the Monitoring Manager, which finally verifies the messages against its execution model based on the automata.

The authors claim that the overhead for monitoring a basic system with two monitors was negligible. The performance characteristics for a larger/more realistic scenario were not discussed. Furthermore, there is no consideration for a trade-off between monitoring coverage and monitoring impact or cost.

### 3.4.15 Monitoring BPEL Assertions

Lazovik et al., 2004 presents a framework for linking assertions representing business rules with WS-BPEL processes. The framework models business processes as a graph of States representing the current state of process execution, and Actions representing business activities that are the transitions between States. Actions are executed on behalf of Roles (Roles are generic businesses, e.g. Travel Agency). Roles are mapped to Providers through service registries. Each business process has a Process Variable which contains all
information on the current state of execution (so the Process Variable for Travel Agency may have information about destinations, hotels etc. from various processes).

Assertions may be made over individual aspects of a composition (e.g. Hotel Price < $300p/n), or an entire composition (e.g. Total cost < $10,000). Assertions may be defined at the business process, role, or provider level. The authors state that any party can publish assertions, although where and how this is achieved is not reported. The assertions and associated user requests are defined in XSRL [30], which is capable of handling Functional and QoS requirements. The XSRL is extended to handle business process flows (choreographies).

![Figure 3.7: Planning and Monitoring Framework [29]](image)

Figure 3.7 shows the Planning and Monitoring Framework for the system. The Monitor is responsible for interleaving the planning and execution of the process. The Planner produces an execution plan for the Monitor, if one exists that meets the user’s goal. The plan takes the form of a sequence of actions to be executed. Upon failure the Monitor will attempt to create a new plan with other available service providers. Once a plan is established, the Monitor passes it to the Executer, which executes the plan and sends any new information (e.g. failures) back to the Monitor for evaluation.

Although this framework involves monitoring, it is targeted at generating an execution plan rather than performing monitoring or configuring a monitoring system. Furthermore, there is no consideration of the performance impact of monitoring.
3.4.16 DEC-based Web Service Monitoring

Rouached and Godart, 2006 presents a method to verify the behaviour of WS compositions by analysing logs of their behaviour using Discrete Event Calculus (DEC). The main focus of the verification is on the order of execution of a Web Services process. The verification is performed by describing the expected properties and the execution log in DEC, and verifying the latter against the former using SPIKE (an automated induction-based theorem prover).

The authors introduce a set of logging techniques for SOAs, which embed process information in SOAP headers, in order for a SOAP intermediary to intercept and log choreography details. Orchestration details are logged via an extended WS-BPEL engine. The logging systems are implemented using bpws4j and log4j.

This solution requires that all Web Services processes are manually described in DEC, and all logs of Web Services process executions are translated to DEC. Since this is performed manually, it could become too complex/error prone for a non-trivial system. Furthermore, the performance impact of the proposed logging techniques is not mentioned by the authors.

3.4.17 Query-based Distributed Monitoring

The authors note the need to monitor and test distributed systems in order to identify faults and measure QoS, and that this monitoring should be able to adapt with the system under observation [61]. The authors claim that monitoring and fault detection “...represents the most involved and costly aspect of designing, implementing, deploying, and operating a widely distributed system”. To reduce this cost, they propose a monitoring technique based on a declarative rule engine, an application logging tool, a generic monitoring tool, and an agent for fault diagnosis. The authors have considered the performance overhead of monitoring, and have designed and benchmarked their system accordingly.

The approach uses an executable data-flow graph with monitoring and diagnostic components to describe systems under evaluation. Since this approach is for a specific system (P2) which is not directly applicable to Web Services or SOA, further implementation details are omitted here.

However, the authors have performed detailed evaluations on the performance impact
of monitoring with their system. The authors have measured an average increase in CPU utilisation of up to 40%, and an average increase in memory consumption of up to 66%, when enabling their monitoring system. The authors also note that CPU consumption grows super-linearly with the rate of probe-based monitoring.

Whilst identifying the need to trade-off between performance and monitoring at runtime, this performance trade-off has not been implemented and is left as future work. Furthermore, whilst these performance tests demonstrate the potential impact of monitoring a distributed system in resource consumption terms, they are not measured in terms of a reduction in delivered QoS.

### 3.4.18 SCALEA-G

The authors identify the need to monitor and analyse the QoS of grid-based services in order to optimise resource allocation for those services [64]. To meet this need, they provide a classification of QoS metrics as well as a framework for monitoring individual services and service interactions.

The proposed QoS taxonomy is shown in Figure 3.8. QoS types are divided into Performance, Dependability, Configuration, Cost, and Custom taxons, each of which (except for Cost and Custom) is sub-divided into specific quality attributes, such as communication time, response time, confidentiality, and reliability.
Figure 3.8: QoS Taxonomy [64]
The proposed monitoring architecture consists of a set of sensors monitoring a set of grid services. Sensors have management (Web Service Distributed Management (WSDM)-enabled Web Services) and information interfaces, allowing for their control and QoS metrics to be retrieved by a QoS Client. Sensors may also observe external services.

The monitoring framework divides monitoring targets into Applications (e.g. service interfaces), Middleware (e.g. SOAP servers), Machines (which host Applications and Middleware), and Network Paths, which join Machines. Each of these resource types is assigned different monitoring techniques. For example, the availability of a Machine is monitored with a ping command, whilst the availability of an Application is monitored with a WS interface. Whilst this is a static set of (Resource Type, Monitor Type, Quality Type) tuples, the set may be extended for use with new resource types.

A set of monitors that provides appropriate coverage for this set of monitored resources is configured using the authors’ SCALEA-G middleware [33]. Each monitor returns monitoring results using this same middleware. This middleware can then detect QoS violations and allow for administrators to correct for faults or poor resource allocation. The middleware also provides an interface for accessing historical QoS records, in order to predict future QoS of service compositions, as well as predict resource usage based on predicted client demand, although the implementation of this is future work.

The proposed framework has been partially prototyped and tested on grid-based systems. However, performance was not tested as these tests considered only the functionality of the framework.

Whilst this work proposes a technique for high-level management of monitoring in distributed systems, it does not consider either monitoring coverage or the performance impacts of monitoring. Rather, it is assumed that an administrator will determine what to monitor and how, and monitoring requirements are not considered. Furthermore, the work uses a fixed QoS taxonomy, rather than a generic QoS model that allows for any QoS aspects to be monitored.

### 3.4.19 ECA-based Web Service Monitoring

The authors identify the need to perform monitoring of component services in WS systems in order to calculate the QoS of composite services and perform service management [66].
To achieve this, they present a QoS monitoring system with a focus on high performance monitoring (that is, monitoring designed for high-throughput systems). The monitoring solution consists of a standard SOA infrastructure extended to explicitly include monitoring, and a technique for high-performance QoS computation.

The authors propose a model-driven method for defining and monitoring QoS in WS systems. The model defines the set of QoS metrics of interest along with rules on how and when those metrics are computed. The model involves integrating monitors into the SOA infrastructure in order to cope with monitoring flexibility, high throughput (and therefore frequent monitored and reported events), complex metric computations such as composite metrics, and metric value persistence (i.e. a database of monitoring records). The authors argue that metrics should be computed and analysed in real-time. The monitoring model allows for a set of ECA rules to be defined, which will be translated into monitors and monitoring actions. For example, an ECA rule may state that “When a service is executed (Event), if the consumer is gold-level (Condition), monitor for Response Time (Action)”.

These rules must be specified manually by a system expert. Once specified, these rules are used to automatically create monitoring instances and set them to monitor appropriate metrics. ECA rules are also used to define what QoS values should be updated based on the output from monitors.

The proposed SOA extension involves the inclusion of three major components:

1. A WS Observation Manager, for allowing users to create observation models (the ECA rules described above);
2. A Metric Computation Engine, for generating monitoring code based on the given observation models, deploying that code, and computing monitoring results; and
3. A QoS Data Service, for storing recorded QoS and monitoring rules.

The Metric Computation Engine translates a set of ECA rules into a state chart that describes which metrics should be computed after which events, and how metrics should be composed. This allows for faster processing, by executing state charts as events occur. The system has been implemented, a small scenario was prototyped as a WS process, and a set of ECA rules were created for this scenario. These rules were automatically translated into state machines, as described above. Testing demonstrated that the monitoring framework...
could handle a large load of up to 2000 events per second. However, the authors did not test the impact of the framework on the monitored system. Details of scalability, or the upper limit of performance were not reported.

Whilst this work focuses on high-performance monitoring, the goal is for monitoring to be able to handle large throughputs with a fast turnaround. The impacts of this monitoring are not discussed by the authors. As such, there is no performance analysis or trade-off.

3.4.20 WSLogA

da Cruz et al., 2004 presents a Web Services utilisation monitoring solution based on interception and logging of SOAP messages by SOAP intermediaries. The architecture, WSLogA, is based on analysis of utilisation (service execution) logs. WSLogA uses both Active (able to modify the SOAP message) and Passive (cannot modify the SOAP message) SOAP intermediaries. The WSLogA architecture is divided into:

- A Composite Web Services Provider, which publishes composite Web Services;
- A Basic Web Services Provider, which provides basic (non-composite) Web Services;
- A Probing Authority Module, which creates and manages Probes, which are Web Services that execute Composite/Basic Web Services for QoS (availability, accessibility, integrity, reliability and basic performance such as response time) testing purposes;
- SOAP Intermediaries, which intercept SOAP messages from Basic Web Services Providers, Composite Web Services Providers, and Proving Authority Modules and output results to the Log Repository;
- An XML Log Repository, which stores the SOAP Intermediaries’ Logs;
- Log Configuration Parameters, which contain probing and logging instructions for SOAP Intermediaries; and
- A Web Server Log Parser, which transforms ASCII logs from the Web Server into WSLogA standard XML logs.
The logging architecture is used to verify SLA-based requirements from consumers, as well as business-level requirements from the service provider. A set of requirement types (e.g. ‘Time taken for service provider to respond’) is mapped to a set of metrics (e.g. ‘Service request time minus service response time’), which are in turn mapped to fields in service execution logs. This allows for automatic verification of requirements by analysing logs, providing those logs include the required fields.

The authors have partially prototyped their technique, and report the performance impacts as ‘negligible’ compared to average network latency. However, the method of monitoring only allows for post-execution analysis, and is targeted at long-term analysis of trends rather than run-time QoS validation. Furthermore, the set of quality types measured is limited, and the implementation of monitors requires custom SOAP intermediaries at each monitoring location, rather than allowing for existing or non-SOAP monitors to be used.

3.5 Monitoring Resources for Scheduling

This section contains reviews of research efforts which present systems for monitoring the local resources of servers that host Web Services. The aim of these techniques is to either have a service provider only accept service requests with SLAs that it knows it can meet, or re-allocate resources so that SLAs will be met.

Netto et al., 2007 presents a scheduler defined to maximise QoS for a utility computing service provider. The scheduler takes advanced reservations for service requests and schedules them in order to maximise QoS of each request’s SLA.

Ranganathan and Dan, 2005 presents a Web Services management approach designed to monitor and dynamically reallocate local system resources for services based on comparing the services’ recently provided QoS to those services’ SLAs. The method takes the performance impact of reallocating resources into account, and is targeted at systems that are computationally intensive (e.g. scientific research and simulation).

The two research efforts in this section present solutions for monitoring local server CPU resources. There is no reason that the service provider could not monitor other resources, such as hard drive space, databases (e.g. ensure that a database server does not become overloaded), human resources (e.g., do not send too many requests to a human
that cannot handle them all), or other services (e.g., do not knowingly overload a helper service). However, this may blur the line between service-level resources and business-level resources.

Ranganathan and Dan, 2005 is unique amongst all works presented here in that it takes into account the cost of reallocating resources on the service provider’s server. This is an important consideration, since a naïve approach may end up spending too much time reallocating resources instead of actually executing service requests.

3.5.1 SLA-based Web Service Reservations

Netto et al., 2007 presents a scheduler designed to maximise QoS for a utility computing service provider. The scheduler takes advanced reservations for service requests and schedules them in order to maximise QoS of each request’s SLA. SLAs are proposed by the service consumer and accepted by the service provider. The SLAs contain elements such as Time (how long the computation will run for), Quantity of Computation, and Price. Service consumers use the SLA to reserve service provider resources in advance, in order to ensure that their computing needs are met.

The SLA constraints provided are:

- Rigid versus Mouldable (i.e., must be performed in one time block, or can be separated into multiple time blocks);
- Timing constraints (start and end time);
- Budget; and
- Computational Resources required (minimum and maximum number of ‘resources’, such as nodes or CPU cycles).

Each time the service provider receives a request for service the scheduler sorts the existing queued service requests and determines if the new request can be accepted. If the new request cannot be accepted, the scheduler returns a list of alternative times that the request can be accepted for. The sorting techniques used are Shuffle (random), First In First Out, Biggest Job First, Least Flexible First (according to start time and deadline), and Earliest Deadline First. Experiments showed the Earliest Deadline First sorting technique as the
technique that produced the greatest resource utilisation (e.g. is most efficient from the service provider’s perspective).

Whilst the scheduling algorithm allows for trading off of deadline requirements for cost in SLAs, the performance impact of the monitoring system is not discussed.

3.5.2 Resource Allocation for Web Services

Ranganathan and Dan, 2005 presents a Web Services management approach designed to monitor and dynamically reallocate local system resources for services based on comparing their current QoS to their SLAs. The method takes the performance impact of reallocating resources into account, and is targeted at systems that are computationally intensive (e.g. scientific research and simulation).

The authors see the need for a system to translate (although the ‘how’ is not reported) high-level QoS descriptions (e.g. “Complete batch processing by 13:00 31/10/08) to low-level resource requirements (e.g. dedicate 5 specific nodes to the task). These high level requirements can be reused, for example if the requester is submitting the same job each week.

The architecture for the proposed system contains the following:

- An Application Service Level Manager, which establishes and monitors SLAs following the WS-Agreement protocol and translates the high-level SLAs into low-level SLAs and estimates resources required (although the translator and resource calculator have not yet been defined, and the authors simply use the average historical process requirements for the same application request in order to estimate required resources);

- A Service Instance Request Scheduler, which handles multiple service executions belonging to a single service request by mapping each new service execution to the original service request and its SLAs;

- A Service Instance Pool Manager, which manages the available resources by allowing or disallowing new service executions or terminating existing service executions in order to maximise QoS or business value;
• A Physical Resource Manager, which allocates and deallocates servers to service instances; and

• A Service Instance Provisioner, which farms a service request to multiple servers.

When a service consumer lodges a request, they must state the relationship between utility (measured in how much they are willing to pay) and response time for the service response.

The performance impact that is considered when reallocating services takes into account many possible (pre-determined) impacts, such as reloading software or libraries, moving services to different nodes, reconfiguring networks, clusters, or file shares.

Whilst this work aims at optimising the total (consumer) utility provided by a service provider, the work is targeted specifically at resource allocation in systems with long-term execution profiles. SLAs are monitored at run-time, however the implementation and performance impact of this monitoring is not reported.

3.6 Summary of Web Service Monitoring Literature

As discussed in this chapter, we have performed a survey of monitoring literature with particular emphasis on monitoring for web services and distributed systems, and dynamic and adaptive monitoring systems, as well as any system, web service-based or otherwise, that takes monitoring impact into account. Appendix C provides a high-level summary of each of these works along with their capabilities according to the following set of criteria:

• The **Technique** used for monitoring, indicating whether monitors were Probe-based\(^3\), Interceptor-based, Eavesdropper-based or a mixture of techniques;

• The **Location** of the monitor(s) (e.g. intermediary message interception, service-side monitoring process or consumer-side gateway);

• The **Abstraction Level** of the approach (e.g. generic technique or implemented framework);

• Whether the monitoring is **Dynamic**, where the monitoring system allows for run-time reconfigurations either manually or automatically, based on results it receives;

---

\(^3\)Chapter 4 provides a definition for these monitoring technique types
• Whether the approach **Balances Monitoring Impacts** with monitoring coverage either for complete or incomplete coverage;

• Who the **Beneficiary** of the monitoring is (for example, service consumer or service provider);

• Who **Performs** the monitoring (for example, service provider, service consumer or third-party intermediary);

• **What** is monitored (for example, services, compositions or processes);

• Whether **Business-level Costs** from monitoring impacts are modelled in the approach;

• Whether a **Bad Proxy** is used, where the system unnecessarily uses an interceptor-type monitor; and

• Whether the **Impact** of the approach is assessed, and if so what that impact is.

Appendix C provides a general review of the monitoring literature that is independent of our research goals. In this chapter, we have focussed on the monitoring literature with respect to our goals described in Chapter 2.

Recall from Chapter 2 that our set of requirements for monitoring and monitoring management are:

1. A generic classification of monitors and monitoring impacts, allowing for trade-offs between monitoring capabilities and their impacts (covered in Section 3.6.1)

2. A method to calculate an optimal monitoring configuration and apply it at design time, including:

   (a) A method to calculate net QoS impacts of a given monitoring configuration (covered in Section 3.6.2)

   (b) A method to determine net business value associated with a given monitoring configuration (covered in Section 3.6.3)

   (c) A method to optimise a monitoring configuration by balancing monitoring coverage with monitoring’s QoS impacts (covered in Section 3.6.4)
3. A method to repeatedly reconfigure service monitoring at run-time to maintain an optimal system state (covered in Section 3.6.5)

4. A method to update knowledge on the impact of monitors at run-time using information from monitoring logs (covered separately in Section 7.4).

Below, we summarise the monitoring literature with respect to these requirements.

3.6.1 Provide a Generic Classification of Monitors

None of the works identified in our survey provides a generic classification of monitors and their impacts.

3.6.2 Calculate Net QoS Impacts of a Monitoring Configuration

Of the 38 works surveyed, 14 have mentioned that minimising the impact of monitoring (whether it be performance, setup cost, or running cost) would be beneficial. Of those, eight have provided measurements of the impact of their monitoring solution. This highlights that whilst minimising the impact of monitoring is identified as valuable, most authors have not described methods to achieve this or even measured the impacts of their monitoring systems.

Baresi and Guinea, 2011, Bertolino et al., 2007, Panahi et al., 2008, Benharref et al., 2009, Narendra et al., 2008, da Cruz et al., 2004, Raimondi et al., 2007 and Zhang et al., 2007 are the only works presented that provided measurements of monitoring impacts.

Panahi et al., 2008 presented a middleware for management and monitoring of Web Services. This is the only work that presents a framework for monitoring that includes a detailed measurement of performance impacts. The authors have identified the need for the management system to be efficient enough to manage and monitor a large number of services without affecting the execution of the underlying business process. The system has been prototyped as a Mule ESB extension. This prototype has been used to compare the performance impact of monitoring. The performance impact determined was between 1–7% for local agents (a type of proxy), and negligible for the service bus (since it is a passive listener). This work provides for manual re-configuration of monitoring systems, with consideration for optimal monitoring configurations with complete monitoring coverage.
Both da Cruz et al., 2004 and Raimondi et al., 2007 presented methods for monitoring using SOAP intermediaries. They have prototyped their respective proposed frameworks and claim insignificant or negligible overheads for performance. However, details of these measurements were not described. Raimondi et al., 2007 presented a method for the verification of service level agreements at run-time using *timed automata*. Monitoring is based on the interception and analysis of SOAP messages by SOAP intermediaries. The performance of the verification algorithm has been considered, and the authors have prototyped the system and reported no “significant overheads”. da Cruz et al., 2004 presented a method for measuring WS usage, based on intercepting SOAP messages and analysing these messages with SOAP intermediaries. The system performance has been considered, and a prototype has been tested. The performance impact of the prototype was reported to be “negligible”, compared to the network latency of target systems.

Baresi and Guinea, 2005, Ludwig et al., 2004, Ranganathan and Dan, 2005 and Keller and Ludwig, 2002 present methods or frameworks capable of monitoring Web Services with consideration for the performance or overhead costs of monitoring. Although each of these discusses the performance impact of their respective monitoring and management systems, none provide information on measurements of the performance impact of their proposed solution.

Overall, none of the works surveyed provides a method to calculate a set of net QoS impacts from a Web Service monitoring configuration. Whilst some impacts are measured, they are typically benchmarks of proposed systems rather than techniques for measuring monitoring impacts on target systems.

### 3.6.3 Determine Business Value of a Monitoring Configuration

Leitner et al., 2010, and Zhang et al., 2008 provide the capability to calculate the business value of monitoring by linking monitoring’s coverage directly to costs that will be incurred from SLAs or other business requirements. However, Leitner et al., 2010 focuses on minimising the occurrence of SLA violations with service re-composition. Zhang et al., 2008 do consider minimisation of monitoring costs, however they do not link those costs to any QoS aspect or SLA.
3.6.4 Balance Monitoring Coverage with Monitoring Impacts to Maximise Business Value


Bertolino et al., 2007 identifies the need to balance monitoring of SLAs with the impact of monitoring, however they present only a high-level, conceptual scheme to achieve this, and consider performance impact from monitoring in terms of resource consumption only.

Of the works that consider balancing monitoring coverage with monitoring impacts, Baresi and Guinea, 2005 and Baresi and Guinea, 2011 are the most advanced, as they trade off generic monitoring levels with performance, whilst most of the other techniques focus on specific costs (e.g. human resource or bandwidth). Baresi and Guinea, 2005 and Baresi and Guinea, 2011 are also the only works that consider disabling monitoring if the cost of monitoring (in terms of impacted QoS) is greater than its benefit. However, neither of these works provide a mechanism to link the benefits or costs of monitoring to SLAs or other business-level attributes.

3.6.5 Maintain an Optimal Monitoring Configuration at Run-time

Abdu et al., 2002, Bai et al., 2009, Baresi and Guinea, 2005, Baresi and Guinea, 2011, Bertolino et al., 2007, Comuzzi and Spanoudakis, 2010, Leitner et al., 2010, Molina-Jimenez et al., 2004, Narendra et al., 2008, Panahi et al., 2008, and Shao et al., 2010 each provide a mechanism to maintain an optimal monitoring configuration at run-time by performing dynamic modification of that monitoring system. Baresi and Guinea, 2005, Bertolino et al., 2007, and Shao et al., 2010 allow for automatic re-configuration of the monitoring system whilst the rest require manual (human) control.

\(^4\)Complete monitoring coverage is when every QoS aspect in the system is monitored regardless of the costs or benefits of performing that monitoring.
The work that is closest to being able to automatically maintain an optimal monitoring configuration at run-time with respect to business value and particularly SLAs is Baresi and Guinea, 2005. Baresi and Guinea, 2005 discussed the performance impact of monitoring, and the solution presented allows for the management of the performance impact of monitoring at run-time. However, this work does not directly consider the business value of monitoring or the value of meeting SLAs.

Whilst not optimising monitoring configurations, Ranganathan and Dan, 2005, Ludwig et al., 2004 and Keller and Ludwig, 2002 each optimised a web service system to maximise business value.

Ranganathan and Dan, 2005 and Ludwig et al., 2004 presented solutions aimed at guaranteeing the quality of service provided for a WS by allocation of resources to Web Services, and systematically accepting WS requests based on the ability to meet SLAs, with consideration for current and expected future load on the Web Services. However, neither work provides a measurement of the impact of their proposed monitoring system or attempts to optimise that monitoring system.

Keller and Ludwig, 2002 presented a framework for monitoring SLAs in web services-based systems with dynamic business agreements. The authors briefly mentioned the impact of monitoring on the performance of the monitored Web Services, but did not measure or minimise the impact of monitors in their proposed system.

3.7 Open Research Issues

As discussed above, there is a wide range of research relating to the monitoring of WS-based and distributed systems. Whilst many authors identify the need to minimise the performance impact or other costs of monitoring, few actually implement techniques to achieve this, and fewer still benchmark their results. Only eight of the works reviewed above consider a trade-off between monitoring benefits and costs, and these are either manual solutions or high-level conceptual frameworks.

None of the works reviewed provide a mechanism to automatically balance the benefits and costs of monitoring at run-time in terms of SLAs. Furthermore, those works that do consider the costs of monitoring either assume monitoring costs are static and can be manually provided, or measure costs in terms of raw resource use of monitors, rather than
linking costs to delivered QoS.

The set of major unsolved problems we have identified for WS monitoring is:

1. Providing a generic classification of WS monitors and measuring the performance impact profiles of each monitoring class;

2. Providing a method to calculate an optimal monitoring configuration, based on costs and benefits from SLAs, business rules, and other sources;

3. Providing a method to maintain an optimal monitoring configuration at run-time, as system requirements, cost and benefits, and delivered QoS change; and

4. Providing a method to measure and continuously refine knowledge of monitoring impacts on a run-time system.

This thesis addresses each of these problems, and reports our approaches in the following chapters. Chapter 4 provides a solution to providing a generic classification of WS monitors and measuring the performance impact profiles of each monitoring class. Chapter 5 provides a mechanism to calculate an optimal monitoring configuration, based on costs and benefits from SLAs, business rules, and other sources. Chapter 6 provides a mechanism to maintain an optimal monitoring configuration at run-time, as system requirements, cost and benefits, and delivered QoS change. Chapter 7 provides a mechanism to measure monitoring impacts on a run-time system.
Chapter 4

Cost of Monitoring

Service Oriented Architecture (SOA) provides for distributed software with loose coupling, user-level composition, and a high level of business support through aspects such as Service Level Agreements (SLAs) and various management standards [47]. Web Services (WSs) represent the dominant means for implementing SOAs. Since Web Services are usually under the control of third parties, their properties, such as response time, are important when considering the overall quality of an application that uses them and are often used to verify SLAs and optimise service compositions or WS infrastructures [47].

Certain service properties (e.g., response time) can only be determined at run-time via monitoring [8]. Monitoring is generally performed by a monitoring agent, a component in a Web Services system that monitors some aspect of that system. For example, a monitoring agent may be a WS proxy-based interceptor that logs all messages that it sends and receives. Monitoring agents may also perform actions based on some initial measurement, such as blocking a message due to a security violation.

A service provider may need to monitor and manage a number of properties of a Web Services system, such as performance (response time, resources consumed, throughput etc.), security (security model, trust in partners, certificate quality, key quality etc.), reliability, and availability [44, 45]. Even in simple scenarios with only one or two of these quality properties being monitored, the cost of monitoring the entire system all the time may be greater than the benefit provided by doing so [8]. For example, using a proxy-based interceptor to intercept and log messages in order to measure response time of a WS may increase the response time of that WS to an unacceptable level under an SLA.
In this case, the service provider may choose to monitor the WS using a different method, or not monitor the WS at all.

Unfortunately, there are cases where service monitoring is mandated by law, contract, or business policy. The specific method of monitoring may also be mandated, leaving no choice to the service provider. Even in these situations, knowing the cost of monitoring is still valuable in order to assign resources to the system or bill any costs of monitoring to external parties.

However, there are situations where the designer of a Web Services monitoring system is able to choose how they meet regulations for monitoring. In these cases, monitoring can be appropriately optimised. Optimisation of a monitoring system may involve selecting exactly what aspects are monitored, at what resolution, at what times, and for what qualities. To achieve such an optimisation, the cost of Web Services monitoring must be determined.

This chapter covers details of our research into quantifying the Quality of Service (QoS) impacts of monitoring WSs. We classify WS monitors, and measure the response time impacts of each monitoring class. We also consider the effects that system configurations and system load have on monitoring impacts. The results of this research provide the basis for our remaining research on monitoring optimisation (Chapters 5 and 6), and automated monitoring impact analysis (Chapter 7). Our survey of WS monitoring techniques in Chapter 3 has identified that monitoring is considered a vital aspect of WS systems. Monitoring allows for run-time management of systems, verification and validation of dynamically generated WS systems (such as systems performing dynamic binding or composition), and validation of SLAs. Our survey in Chapter 3 has also identified that monitoring can have a significant impact on delivered QoS, depending on the monitoring techniques used and the system to which monitoring is applied. However, all of the WS monitoring impacts that are reported in the literature report the impacts of specific WS monitoring systems. Instead, we are interested in discovering the relative impacts of different, generic monitor classes and configurations, for which we have encountered no existing work. For example, we have encountered no existing studies that answer the question “Is the performance impact of a Simple Object Access Protocol (SOAP) interceptor greater than the performance impact of software level eavesdropping?”. As such, it is difficult to
perform a direct comparison between monitoring techniques without implementation and testing.

To enable a direct comparison between monitoring techniques, we present a generic WS monitoring classification, based on which a set of prototype systems have been implemented. We have used these prototypes to conduct a series of experiments to measure and quantify the performance impact of monitoring a WS system based on our new classification of monitoring techniques, described in Section 4.1. The following sections describe these experiments and the results obtained from them. For our experiments, we have focused our attention on response time. Response time was chosen as it is a common measure of performance, is a requirement in many SLAs\(^1\), and is easily measurable and quantifiable. Impacts on QoSs other than response time are considered in future chapters, and are included in experiments as well as a case study evaluation in Chapter 8.

Section 4.1 describes our monitor classification. In Section 4.2 we describe existing efforts in measuring monitoring impacts. Section 4.3 contains a detailed design of our experiments for measuring monitoring impacts. Section 4.4 describes the results of those experiments. Section 4.5 discusses our experiment results and their limitations.

### 4.1 Monitor Classification

The classification depicted in Figure 4.1 is our generic monitoring classification based on communication paths between monitors and services. The classification consists of probe, interceptor and eavesdropper classes of monitors (highlighted with grey backgrounds in Figure 4.1), each of which is described below.

#### 4.1.1 Probe-based Monitors

Probes include all methods of monitoring that generate their own messages or other traffic intended for the target system. This may include simulated consumer messages and invocations of test procedures. Probes are unique in their ability to generate monitored events. That is, interceptor and eavesdropper classes of monitors can only observe events such as service invocations, whereas probes can generate and observe those events.

\(^1\)See Appendix A for examples.
This means that QoS and functionality of Web Services can be measured either off-line, or when no consumers are using a service. **Probes** are also commonly used by external monitoring agents to measure QoS aspects such as availability from a trusted, third-party perspective[15, 38]. **Probes** are expected to impact the monitored WS system in the following common ways:

- Reducing *performance* by consuming finite processing resources (CPU and RAM, etc.) on the service’s host by executing the service and effectively increasing the service load;

- Reducing *performance* by consuming finite processing resources on the service’s host by performing analysis of monitoring results (i.e., when the probe and service are collocated);

- Reducing *performance* by consuming finite network bandwidth when communicating with the service (i.e., when the probe and service are not collocated);

- Reducing *reliability* by consuming finite processing and network bandwidth resources, causing potential overloads such as 100% CPU usage, significant RAM paging, and dropped network packets.

While this is not an exhaustive list, it reflects what we consider the most likely impacts from using **probes** to monitor services.
4.1.2 Interceptor-based Monitors

**Interceptors** include all methods of monitoring that involve implicit or explicit interception of message flow between a source and intended recipient, and implicit or explicit modification of execution flow. This includes hardware proxies that intercept network traffic, hardware and software firewalls, wrappers for virtual machines or run-spaces that intercept communications as software executes, software aspects that modify code execution, and software proxies that intercept high-level messages such as SOAP packets and database commands.

**Interceptors** are unique in that they are able to halt, delay and modify communication and execution based on the results of monitoring. For example, network traffic may be redirected through a software firewall that analyses and blocks messages with unencrypted, private information. Due to their unique ability, **interceptors** are commonly used for this type of firewalling. **Interceptors** are also commonly used in WS-based systems to monitor processes such as those defined in Web Services Business Process Execution Language (WS-BPEL), by intercepting SOAP packets at SOAP servers or between services and their consumers [35, 8, 7, 22]. **Interceptors** are expected to impact the monitored WS system in the following common ways:

- Reducing *performance* by lengthening communication and execution paths;

- Reducing *performance* by halting communication for analysis (when performing firewalling);

- Reducing *performance* by consuming finite processing resources on the service’s host by analysing monitored traffic (requires the interceptor and service to be collocated);

- Reducing *reliability* by consuming finite processing and network bandwidth resources, causing potential overloads such as 100% CPU time, significant RAM paging, and dropped network packets; and

- Reducing *reliability* by intercepting and corrupting communication or code flow.

Note that this is not an exhaustive list, but reflects what we consider the most likely impacts from using **interceptors** to monitor services. It is important to note that the
impacts from using interceptors may vary significantly depending on whether the interceptor performs a firewalling action by halting messages for analysis.

4.1.3 Eavesdropper-based Monitors

Eavesdroppers include all methods of monitoring that passively capture existing communications between the consumer and provider. This may include methods such as network level eavesdropping by reading logs on routers and software level eavesdropping by reading logs or listening to events from services, middleware, or operating systems. Eavesdroppers include those monitors that sit on a communications path and record or duplicate messages, as depicted in Figure 4.1. Eavesdroppers also include those monitors that receive messages from a heartbeat or broadcast. Eavesdroppers are commonly used to record logs of system events, measure QoS, and analyse interactions between services in service compositions [9, 15, 50]. Eavesdroppers are expected to impact the monitored WS system in the following common ways:

- Reducing performance by consuming finite processing resources on the service’s host by analysing monitored events (i.e., when the eavesdropper and service are collocated); and

- Reducing reliability by consuming finite processing resources, causing potential overloads such as 100% CPU usage, significant RAM paging, and dropped network packets (i.e., when the eavesdropper and service are collocated).

We consider eavesdropper-based monitors to be likely to have the least performance impact on monitored Web Services, due to having no direct interaction with either service-consumer communications, or services themselves. However, eavesdroppers are limited in that they cannot generate monitoring results as probes do, and cannot halt or modify communications as interceptors do.

4.1.4 Summary of Monitor Classification

We have classified monitor types according to their method of gathering information, and described the typical uses, unique properties, and likely impacts of each monitoring class. The monitoring classification is designed to be valid for all Open Systems Interconnection
(OSI) or other technology stack levels. For example, a generic interceptor monitor may be realised using a hardware proxy such as a network router, or a software proxy such as a modification to a SOAP server instance.

Other methods of monitoring are available, such as instrumentation of WS or client code, and instrumentation of virtual machines or runtimes. The performance of instrumentation-based methods is not measured here, since these methods require access to source code or virtual machines, which may not always be available. These instrumentation based methods are also restricted in that instrumentation-based monitors can only be placed where a virtual machine or application exists, whereas network level interceptors may be placed anywhere along communication paths. Some of the results of interceptor-based monitoring testing that are presented in this chapter may however be applied to instrumentation-based monitoring. For example, a major cause of delay for a blocking interceptor is the action that is taken by the interceptor, rather than the overhead of hosting the interceptor itself. In this case, an instrumentation-based interceptor will yield a similar performance impact to other types of interceptors such as those used in our experiments. Additionally, the impact of probe-based monitoring is not included in these experiments, as the impact of probe-based monitoring is not expected to be consistent across systems, due to the many possible types of probes that may have impacts ranging from negligible to extreme. For example, a probe may request the current processor usage, or it may stress test a server.

4.2 Existing Efforts in Quantifying Monitoring Impact

We have provided a survey of Web Service and distributed software monitoring techniques in Chapter 3. For this survey, 39 research efforts were reviewed, of which only five provided a quantified impact assessment of monitoring [6, 9, 12, 46, 61]. Three works stated the impact as “negligible” due to use of eavesdropping or existing proxies [20, 22, 51]. Whilst the impacts of monitoring have been determined to be significant for many systems, these results show that in general, designers of distributed monitoring systems have placed little importance on modelling performance impacts. Those authors that did measure Web Service monitoring impacts only measured the impacts of their specific monitoring solutions. That is, none of the reviewed research efforts considered the performance impacts of Web
Service monitoring in a generic way.

Bai et al., 2009 reported a Web Service monitoring impact of “less than 5%”. This impact was measured as the overhead of performing monitoring using a prototype of the authors’ service monitoring system. Monitoring was performed using an eavesdropper-type monitor that monitored event broadcasts from an existing Web Service proxy.

Baresi and Guinea, 2011 reported a Web Service monitoring impact of “10–100%”. This impact was measured as an increase of response time, relative to the response time of an unmonitored service. The variations in impact were due to different levels of monitoring analysis being performed on WS-BPEL servers, which were observed using eavesdropper monitors.

Benharref et al., 2009 reported a Web Service monitoring impact of “1–5%”. This impact was measured as the increase in CPU and RAM usage when monitoring was enabled, where monitors were eavesdropper monitors that observed and analysed service interactions for faults.

Panahi et al., 2008 reported a Web Service monitoring impact of “1–7%”. This impact was measured as the increase in response time when monitoring was enabled. Monitors observed WS-BPEL processes for QoS, and could be configured to monitor as eavesdroppers or probes. Probes generated an impact of up to 7%, whilst the impact of eavesdroppers was reported to be negligible.

Singh et al., 2006 reported a Web Service monitoring impact of “up to 40%”. This impact was measured as the increase in CPU utilisation when monitoring was enabled, compared to no monitoring. Monitors were used to detect faults and measure QoS, and consisted of probe-based and eavesdropper-based monitors. However it was the probe-based monitors that had the larger impact, as well as a faster growth in impact as monitoring frequency increased.

da Cruz et al., 2004, Gan et al., 2007 and Raimondi et al., 2007 all report the monitoring impacts for their systems as ‘negligible’. da Cruz et al., 2004 uses existing SOAP intermediaries to log SOAP packets sent between Web Service, and report the impact on response time as negligible. Gan et al., 2007 use eavesdroppers to monitor Web Service conversations, and report the overhead of using their monitoring technique for a small prototype system as negligible. Raimondi et al., 2007 uses eavesdroppers to
monitor Web Services for temporal requirements, and reported the overhead of using their monitoring technique for a small prototype system as negligible.

Whilst these authors have identified the need to quantify and minimise the impacts of Web Service monitoring, they have only measured impacts in terms of their own, specific Web Service monitoring systems. There has been no work that considers the generic properties of Web Service monitors, and attempts to discover the relative difference in impacts between different generic Web Service monitoring configurations.

4.3 Experiment Design

In this section, we present details of experiments designed to measure the performance impact of Web Services monitoring. The purpose of these experiments is to determine the relative response time impacts of interceptor-based and eavesdropper-based WS monitors. The experiments are also designed to measure the effect that monitor location, monitoring analysis, and amount of service load have on monitoring impacts.

4.3.1 Monitoring Parameters

To achieve our objectives, we intend to investigate the impacts of the following parameters for the monitoring system:

(1) The use of a WS proxy-based interceptor;

   (a) The type of action(s) of the WS interceptor;
   
   (b) The location of the WS interceptor;
   
   (c) Whether the WS interceptor is blocking or non-blocking; and

(2) The use of an eavesdropper (an eavesdropping, software-based web service monitoring system).

The first monitoring parameter (1) will be used to determine the performance impact of using an interceptor, compared to directly invoking a WS. Once this impact is established, any processing that is performed at the interceptor can be compared with this baseline response time. Parameter (1.a) will be used to determine the performance impact of different actions that an interceptor takes, such as logging, or performing some
CPU/RAM intensive task, which may be likened to eXtensible Markup Language (XML) serialisation/de-serialisation or encryption/decryption. Parameter (1.b) will be used to determine the impact of the interceptor’s location. The location may be either on the server hosting the Web Service, or on a remote, dedicated computer. Parameter (1.c) will be used to determine the difference between blocking interceptors and non-blocking interceptors. Parameter (2) will be used to determine the impact of using a passive eavesdropper to monitor a Web Service.

Each of these parameters will be tested with a range of client load levels.

4.3.2 Monitoring Configurations

We have developed a prototype system and monitors in order to measure the effect of each of the parameters identified in Section 4.3.1. We have developed both interceptor and eavesdropper based monitors. Three variants of the interceptor monitor were created. First, a basic, empty interceptor was used as a baseline for the cost of interceptor-based monitoring. Second, a variant wherein the interceptor has a constant delay was created in order to establish the cost of a delay in the interceptor. Third, a variant wherein the interceptor performed processing that was equivalent to the processing performed by the monitored Web Service was created in order to establish the cost of an interceptor that is performing heavy processing. This variant has ‘blocking’ and ‘non-blocking’ forms. The blocking version receives a message, performs the required processing, and then forwards the message. Conversely, the non-blocking version receives a message, forwards the message, and then performs the required processing.

Only one variant of the eavesdropper monitor was developed, which uses a software-level packet interceptor to log and analyse WS traffic. This second monitoring configuration (2) was used to determine the impact of monitoring a Web Service system without directly modifying that system. In this case, eavesdropper-based monitoring was achieved with software instrumentation of all TCP/IP communications using Wireshark [28].

All of the monitoring impact experiments were measured using 10, 20, 30 and 40 simultaneous clients, in order to determine the impacts of monitoring as load varies.

In summary, the configurations we have tested are:
(1) Location of interceptor
   (a) Nowhere - direct invocation (Figure 4.2, Type 1)
   (b) Local (Figure 4.2, Type 2)
   (c) On a dedicated node (Figure 4.2, Type 3)

(2) Actions of interceptor
   (a) None (Empty)
   (b) Static delay
   (c) Relative load
      (i) Blocking
      (ii) Non-Blocking

(3) Use of an eavesdropper (Figure 4.2, Type 4)

Configuration (1) was used to test parameter (1.b) from Section 4.3.1, configuration (2) was used to test parameters (1.a) and (1.c), and configuration (3) was used to test software-based eavesdropping (parameter (2)).
4.3.3 System Configuration

The system configuration involved the use of a single Web Service, a Web Service client load generator, and a Web Service intercepting monitor. Both a blocking interceptor and a non-blocking interceptor were developed.

The Web Service has a positive natural number as its only input parameter, and returns π to that number of decimal places, as a string. For example, executing pi_service(3) returns “3.142.” The computation of π is based on the Gauss-Legendre algorithm. Through this computation, this service stresses the RAM and CPU of the hosting server, and allows for varying degrees of load based on the input parameter. Apart from this flexibility, this method of load-generation was chosen because it is easily repeatable, and can provide consistent results. This method of load generation generates a relatively high level of server-side processing per client request, which is useful since the server’s processing capacity can be stressed without stressing other elements in the test bed, such as the client system or network. Since Web Services are designed as interfaces to existing systems, stressing these aspects of the system are realistic (e.g., stressing of disk I/O would more likely represent the behaviour of an application that a web service is the interface for). The Web Service was created in Java/J2EE and hosted on an Apache/Tomcat server.

The Web Service intercepting monitor is a simple redirection service. The interceptor was modified to perform various tasks such as sleeping (artificial static delay), and computing π locally (artificial relative load). The artificial static delay allows for the measurement of change in response time due to a known and static delay. This would answer the question “what is the response time impact of a 5 second delay in a Web Services interceptor-based monitor?” The artificial relative load allows a load to be put on the monitor that is equivalent to the load on the server. This allows for a simple baseline measurement of the impact of monitoring with a level of processing equivalent to that of the Web Service itself. The Web Service interceptor was created in Java/J2EE and also hosted on an Apache/Tomcat server.

The Web Service client load generator invokes the pi_service method on the Web Service and measures the total time taken (response time) from when the service is invoked by the client until the result is returned to the client. Apart from the number of places for which to compute π, the client application takes as input num_threads, the
number of threads (simultaneous clients) to execute. Modifying the number of simultaneous clients allows for the simulation of varying loads on the servers hosting the Web Service and monitors. The client simultaneously invokes the Web Service `num_threads` times and returns the average of the response times for all threads.

For eavesdropping, Wireshark was used to monitor the IP traffic for each node’s network interface. Response times were measured with Wireshark running on one, two, or three of the nodes in order to determine if there was any impact.

Figure 4.2 shows the experiment configurations that were used to exercise the required scenarios. Rectangles with rounded corners represent node boundaries. Rectangles represent services or processes, and arrows represent communication links.

Type 1 in Figure 4.2 shows the baseline system, with direct invocation of a WS (no interceptor). Type 2 shows an interceptor co-located with the WS, Type 3 shows the interceptor on a separate server to the WS, and Type 4 shows eavesdroppers placed with the service, with the client, and on a proxy server.

Each result presented is the average of at least 15 test runs (more tests were run to increase confidence, where required) at 10, 20, 30, and 40 threads and 10,000 digits of π. Each average has vertical error bars showing the standard error at 95% confidence for that value. These error bars demonstrate the separation of the average response times.

4.3.4 Platform Details

Table 4.1 presents the platform details for the experiment. Each node involved in the experiment was running Microsoft Windows XP SP3. The experiments were run on a dedicated LAN with no other traffic. All addressing was via IP (no DNS lookups). Apache/Tomcat was used for hosting the Web Services. The network was never under more than 5% load during tests. This eliminates the chance that the network was a ‘bottleneck’ in the system. None of the systems involved in the experiment required the use of swap space during any test.

The systems selected for this benchmark cannot be representative of every possible combination of client, server, and interceptor that is used. However, for our experiments either communications paths are lengthened by using proxies, communication times are increased by halting messages for analysis, or performance is impacted due to fixed resources
being consumed by monitors performing analysis. As such, the comparative results from these experiments should scale to any systems of similar architecture. That is, we expect that running the same experiments on a group of high performance servers would yield faster response times, but the relative difference between response times for different monitoring methods would remain.

### 4.4 Experiment Results

Using the systems and configurations described in the previous section, we have conducted a wide range of performance tests to establish the effect of the various monitoring system parameters. The following sections present the results of the experiments under the most representative scenarios.

#### 4.4.1 Effect of Interceptor Location

Figure 4.3 shows the results from testing the system using no interceptor, an empty interceptor on the same server as the WS, and an empty interceptor on a separate node (Node 2). These results show the impact from using a WS interceptor for monitoring (compared to no monitoring), as well as the comparative impact of the interceptor based on its location. For these tests, the interceptor monitors only forwarded received messages, and performed no blocking, sleeping or analysis. This allows for the performance impact of hosting an interceptor monitor to be established independently of the impact of performing analysis or firewalling.

The results in Figure 4.3 show that direct invocation of the WS was 8% slower than invoking the WS through an interceptor on the server. This counter intuitive result
Figure 4.3: Effect of Proxy-based Interceptor on Response Time
may be explained by the fact that having an extra **interceptor** on the **server** doubles the buffers available on the serving side, reducing the number of rejected connections. This has been confirmed by using Wireshark to count the number of rejected connections for both systems. Our experiments showed that the system not using an interceptor had approximately 40% more rejected connections. Similar results have been reported in Bressler, 2004. These results also show that moving an **empty interceptor** from being collocated with the **service** to another **node** increases the **response time** by approximately 8%.

### 4.4.2 Static Delay at Interceptor

Figure 4.4 shows the results from testing with **empty interceptors**, and with **interceptors** with an inbuilt 5-second delay, on both the **server** and a **dedicated node**, respectively. This allows for a comparison of the impacts of a **blocking interceptor** on a **server** and a **dedicated node**, as well as a comparison of the impacts of **blocking interceptors** and **empty interceptors**.

**Interceptors** on the **server** and **Node 2** with a 5-second delay averaged approximately 14 seconds slower than **empty interceptors** with the same configurations. This shows that the impact of a delay is not a simple arithmetic function. For example, the **response time** of an **empty interceptor** minus the **response time** of an **interceptor** that sleeps for 5 seconds before returning the result is 14 seconds, not 5 seconds. Moving the **interceptor** with a static delay from the WS **server** to a separate **node** increased **response time** by approximately 10%.

### 4.4.3 Processing in Interceptor Located at Server

Figure 4.5 shows the results from testing an **interceptor** that performs a processing-intensive task on the **server**. The results are for **blocking interceptors** and **non-blocking interceptors**, respectively. This figure allows for a comparison of the impacts of **blocking interceptors** and **non-blocking interceptors** located on the server.

As expected, the **blocking interceptor** caused longer **response times** than the **non-blocking interceptor**, by approximately 40%. This difference in the impact on **response time** increased as load increased, due to the number of simultaneous clients.
Figure 4.4: 5000ms Blocking Delay at Proxy-based Interceptor
Figure 4.5: CPU/RAM Intensive Processing at Interceptor on Server
The results also show that the impact of a non-blocking interceptor on response time in this scenario was approximately 30%. Once again, this impact increased as the load on the service increased. The relative impact between the blocking interceptor and non-blocking interceptor has reduced and almost stabilised at around 20 threads. This is to be expected, since once the local resources are saturated due to high load, any additional effect of non-blocking processing at the interceptor is diminished.

4.4.4 Processing in Interceptor Located at Node 2

Figure 4.6 shows the results from testing an interceptor that is performing a processing-intensive task on a second, dedicated node (Node 2). Similar to the previous section, the results are for configurations of the interceptor as blocking and non-blocking, respectively.

As per the previous experiment with the interceptor collocated with the service, the
blocking interceptor caused approximately 40% greater response times compared to the non-blocking interceptor. Once again, this difference in the impact on response time increased as the load (number of simultaneous threads) increased. The results in Figure 4.6 also show that the impact of a non-blocking interceptor on response time in this scenario was approximately 5%.

Figure 4.6 also shows the impact of a Blocking interceptor on a second, dedicated node rising sharply away from the impact of a non-blocking interceptor. In this case, the difference increases at 40 clients due to the CPUs on the interceptor host becoming saturated, however the trend is already detectable from 10 to 30 clients.

Overall, the results on intercepting monitoring impacts show that the main impact comes from performing blocking actions on messages, and that for monitoring performing work equivalent to the service being monitored, response times increase by an average 40%, regardless of the location of the interceptor.

4.4.5 Eavesdropping

Figure 4.7 shows the results from testing the effect of an eavesdropping software-level monitor on the system. The results are the average for the system configured with and without a dedicated proxy server, as adding a proxy server did not cause a change in the average response times. The results are for each scenario run with between 10 and 40 threads.

As can be seen in Figure 4.7, eavesdropping on the system has no measurable negative effect on response time. As such, these results show that it is possible to eavesdrop on a Web Service without negatively impacting the response time of that Web Service.

4.5 Discussion

By analysing the results from monitoring using eavesdropping, it can be seen that it is possible to eavesdrop on a Web Service without significantly affecting the response time of that Web Service. Conversely, by analysing the results for monitoring using an interceptor, it can be seen that monitoring using an interceptor which performs some computation can have a significant negative impact on response time.
Figure 4.7: Eavesdropping vs. No Eavesdropping
The eavesdropping method of monitoring has limited capability however. Firstly, the eavesdropper may not understand captured traffic if that traffic is encrypted. Secondly, as mentioned in Section 4.1.3, eavesdropper-based monitors are unable to generate requests as probes do, and unable to perform firewalling actions as interceptors can. Otherwise, our results show that intercepting monitors should only be used when they are required to perform a blocking function.

The results for testing the effect of a computation and memory intensive interceptor show that the impact from an interceptor performing work (in this case computing $\pi$) may be reduced by over 20% by moving the interceptor from the server hosting the WS to another node. Typical usage for an interceptor would be for logging messages or duplicating and redirecting messages based on content. The interceptors in our scenario had a heavier processing load than would be expected of interceptors performing simple logging or trafficking actions, since they were computing $\pi$. The interceptors were designed this way so that their processing was equivalent to that performed by the Web Service being monitored. This equivalent processing load is realistic in many scenarios, where the Web Service is simply performing some database operation or executing a remote process.

Apart from the impacts of an interceptor-based monitor compared to an eavesdropping monitor, the impacts of a blocking monitor varied significantly, compared to the impacts of a non-blocking monitor. The results show that performing a blocking function with an interceptor was approximately 40% slower than performing a non-blocking function. This is a significant result, and demonstrates that a small design choice in a WS monitor can have a large impact on performance. This impact of 40% on response time occurred for an interceptor on both the server hosting the WS and a dedicated server. This demonstrates that the impact cannot be reduced by simply relocating the interceptor to a dedicated server.

Processing load is an important factor for designers of Web Services systems to consider when designing systems for meeting SLAs. Designers must trade off between the response times provided and the number of servers required.

For each of the sets of results provided in Section 4.4, the response time is given as an average over all numbers of simultaneous clients. For example, the response time for
10 simultaneous clients is directly comparable to the response time for 20 simultaneous clients, since each of these times is an average over all client invocations. Knowing this, it can be clearly seen that as the number of simultaneous clients goes up, so does the average response time. This effect occurred in all test scenarios that were executed.

4.5.1 Limitations

The results presented were based on experiments that stressed the processors of each node in the system. It is important to note that other benchmarks may stress other aspects of a system such as the network, which may yield different results. Furthermore, we have performed these experiments using Java/J2EE with Apache/Tomcat. Whilst we expect that the results will translate to other Web Service implementations and servers, this cannot be guaranteed.

4.6 Summary

We have identified no work that quantifies the impact of various types of Web Services monitoring systems on the quality of service provided by those Web Services. In order to fill this gap, we have conducted a series of experiments in order to quantify and assess the impact of monitoring on Web Services in typical Web Service scenarios. We used response time as a target QoS to measure for impacts, as response time is easy to measure and commonly used as the basis for requirements in SLAs (as demonstrated by the SLA review in Appendix A). These experiments provided three key results: firstly, it is possible to eavesdrop on a Web Service with no measurable negative impact on response time. Secondly, a blocking monitor may significantly increase the response time of a Web Service when compared with a non-blocking monitor. Thirdly, the location of the interceptor monitor has an impact on the response time of the Web Service being monitored.

The experiments also demonstrated that a Web Service monitoring system can incur significant costs in the form of a reduction in QoS provided by a Web Service depending on the implemented monitoring methods (type and location).

These results provide the rationale to investigate strategies to optimise a Web Service
monitoring system at design-time and run-time, as described in Chapters 5 and 6, respectively. They consider balancing the impact of monitoring as discovered in this chapter against the requirements or benefits of monitoring using an optimisation algorithm.
Chapter 5

Design-Time Monitoring Optimisation

As discussed in Chapter 1, there is a need for monitoring Web Service-based systems at run-time. This need for monitoring comes from Web Service-based systems changing at run-time, and from the need for Web Service consumers to know what Quality of Service (QoS) aspects they should expect when invoking services from third party service suppliers. In general, monitoring of Web Services is required in order for service consumers and service providers to measure various quality aspects of Web Service systems. Service providers use the results of monitoring for the management of the Web Services and their supporting infrastructure.

There are numerous properties that one may wish to monitor in a service-oriented system, including performance-related properties, which include quality types such as response time, resource consumption and throughput; security-related properties, which include quality types such as the encryption method used, certificate quality, and key quality; and reliability-related properties, which include quality types such as availability, functional accuracy and correctness[44].

Run-time monitoring and management can increase confidence in the quality of software, and allow for increased quality of software due to an increase in the number of faults that are repaired after being detected. However, monitoring comes with a cost in terms of an impact on delivered QoS, as discussed in Chapter 4. Since Service Level Agreements
(SLAs) may have strict requirements on delivered QoS, the cost of these monitoring impacts may outweigh the benefit of monitoring. Even in simple scenarios with only one or two quality types being monitored, the expense of monitoring the entire system all of the time may be greater than the benefit provided by doing so [9, 60].

A solution that allows parties interested in monitoring the system to select exactly what aspects are monitored, at what resolution, and for what qualities, would allow users of monitoring services to obtain better value from the monitoring system by optimising the configuration of their Web Service monitors at design-time and run-time. This chapter provides the details of a method developed to optimise the configuration of a suite of Web Service monitors at design-time, by trading off monitoring coverage and monitoring impacts.

Section 5.1 provides a description of the optimisation problem, based on the previously introduced iTravel scenario. Section 5.2 provides a description of our modelling of the optimisation problem. Section 5.3 describes our optimisation methodology. Section 5.4 describes the application of our optimisation technique to the iTravel system. Section 5.5 describes a set of tools used to randomly generate problems sets for evaluating our optimisation technique. Section 5.6 evaluates the effectiveness of our optimisation technique.

5.1 The Monitoring Optimisation Problem

Chapter 2 introduced a scenario involving a company, iTravel, that provides Web Services for the travel industry. iTravel aggregates and provides information and booking facilities from various sources for flight times and costs. This information is then provided to travel agents through Web Services.

iTravel offers an SLA to its consumers that provides guarantees on delivered QoS, and includes monetary penalties for SLA breaches. iTravel also has requirements from local laws and corporate governance that require the security and privacy of their Web Services to be monitored. Due to these requirements, it is beneficial for iTravel to monitor both functional and non-functional properties of their Web Services.

iTravel has requirements on both monitoring and delivered QoS. Since monitoring may reduce delivered QoS, iTravel must ensure that Service Level Objectives (SLOs) are
not breached due to excessive monitoring. To achieve this, iTavel must strike a balance between meeting its requirements for monitoring and meeting its requirements for delivered QoS levels. Since we assume that iTavel has a fixed set of resources (i.e. iTavel cannot just increase QoS by adding more resources), iTavel must configure their monitors to achieve as much benefit from monitoring\(^1\) as possible without causing SLA violations.

For example, iTavel may monitor the security of its book_flight service in order to meet corporate governance requirements for monitoring security. However, this monitoring may reduce the response time of book_flight and cause SLOs to be breached, causing iTavel to have to refund account fees to its customers. In this case, iTavel should weigh the benefits of monitoring book_flight’s security with the costs of not meeting book_flight’s response time requirements, and configure monitoring accordingly. This is just one monitoring requirement and one QoS requirement. In order to configure monitoring for the entire iTavel system, all monitoring and QoS requirements must be considered simultaneously, and the configuration of each monitor in the iTavel system must be selected whilst considering the total impact of all monitoring in the system.

In summary, the optimisation problem is to find a monitoring configuration that gives a service provider maximum value or utility in terms of monitoring coverage (benefits) and QoS impact (costs).

None of the published work from the literature review in Chapter 3 provides a method for monitoring and management of a Web Services system with a trade-off for the the net costs and benefits of monitoring a Web Service system, as described above.

### 5.2 A Model of the Monitoring Optimisation Problem

Figure 5.1 shows an Entity Relationship Diagram (ERD) that describes the modelling of our optimisation problem. The diagram shows entities that are explicitly included in our model with white backgrounds. Entities with grey backgrounds are considered but not explicitly modelled. The ERD shows the relationships between core entities in a Web Service system (Monitors, Services, Service Providers, Consumers and SLAs), as well as the relationships between those entities.

\(^1\)Note that we assume there is no cost associated with monitoring other than QoS impacts on monitored services. For example, we do not consider the power costs due to performing CPU-intensive monitoring.
Figure 5.1: Optimisation Entity Relationship Diagram
Working from the Service entity, a Service Has a set of Ideal Quality Levels. These describe the Value of Quality levels (composed of a Unit such as milliseconds and a Type such as response time) that are delivered by that Service under ideal conditions of no external load and no monitoring impacts.

Services are Used by Consumers, who Accept SLAs that are Offered by a Service Provider, and are Composed of a set of Requirements. Requirements may also come from other, non-consumer Stakeholders. For example, requirements may come from corporate governance and be related to local laws and regulations. Each Requirement Imposes a Penalty on the Service Provider for non-compliance. For example, a local law may require all transactions to be monitored for privacy regulations, with a penalty of $10,000 per breach.

Requirements are On a Service’s Delivered Final Quality Levels. When a Service is Used by a Consumer, the Service Delivers a set of Final Quality Levels. Each of these Final Quality Levels is a Percent Of that Service’s Ideal Quality Level For that Quality. If a Final Quality Level is the same or higher than that required (and monitored to demonstrate this), then the requirement is met, and no Penalty is Imposed.

As stated above, a requirement must be monitored in order for it to be shown to be met. Each Monitor Has a set of Monitoring Capabilities. Each Monitoring Capability has the Ability to Measure one Final Quality Level for one Service. For example, the Generic Eavesdropper can monitor response time of the flight_info service (one monitoring capability) and book_flight service (another monitoring capability) in the iTrip scenario. Each Monitoring Capability has a set of possible Monitoring Levels At which it can be set. A Monitoring Setting Instantiates a Monitoring Capability by setting that Monitoring Capabilities’ Monitoring Level At a particular value. When a Monitoring Setting is used to Measure a Final Quality Level, that Monitoring Setting Evinces any Requirements on that Final Quality Level.

Each Monitoring Setting in the system Has 0 or more Impacts On delivered Final Quality Levels. For example, stopping a message and analysing it for security breaches may impact a final quality level (such as response time) of the service sending that message. These Monitoring Setting Impacts occur only when a Monitoring Setting Measures
a Final Quality Level. That is, if a monitor is enabled but performs no actions, it has no Monitoring Setting Impacts. Conversely, each Monitor Has a set of overhead Impacts On 0 or more Final Quality Levels. These impacts occur whenever a monitor is enabled, and are independent of the actions that monitor takes. For example, a proxy-based monitor that redirects traffic may cause response time to increase, even if the monitor does not analyse that redirected traffic.

A Monitoring Configuration is a Complete Set of Monitoring Settings. That is, a Monitoring Configuration describes a Monitoring Level for every Monitoring Capability in the system. Two basic examples of Monitoring Configurations are: (1) having all monitoring settings fully enabled; and (2) all monitoring settings fully disabled. Each Monitoring Configuration has a set of Monitoring Configuration Impacts describing the Sum of Impacts from that Monitoring Configuration. This Sum of Impacts is the sum of all monitoring overhead and monitoring setting impacts for the given monitoring configuration, over all Final Quality Levels.

5.3 Optimisation Methodology

Section 5.1 highlights the need to optimise the configuration of deployed web service monitors, so that the impact the monitors have on delivered QoS can be minimised whilst valuable monitoring requirements are met, as well as our modelling of the optimisation problem space.

To achieve our optimisation goal, information about a system’s services, monitors, and requirements is needed. Information required on services includes what services exist and what QoS levels they provide. Information required on monitors includes what monitors exist, what QoS aspects they can monitor, the levels of monitoring available\(^2\), and the impacts that monitoring has on each QoS in the system. Information needed on requirements includes all SLAs, corporate governance, and other requirements on the Web Service system. These requirements should be formulated in terms of pairs of a QoS level required, and a monitoring level required to verify that QoS was delivered. For example, one QoS and monitoring requirement may be (“response time of the book_-

\(^2\)For example, an interceptor may have 5 monitoring levels representing 5 different monitoring levels between 0% and 100%
flight service must be under 10 seconds, and must be monitored with a sampling rate of at least 50% to verify this). Each of these requirements is then linked to penalties for requirement breaches. For example, one penalty may be “50% refund to consumers for response time over 10 seconds”, and another a “$5,000 fine to government for breaching privacy act”.

Figure 5.2 shows a framework illustrating our approach to solving the monitoring optimisation problem, annotated with the ordering of activities of the optimisation process. In step 1, the set of deployed monitors is identified based on IT records, and then the Enumerator generates all the possible monitoring configurations for this set of monitors. In step 2, the performance impacts of each monitor are identified from IT Records or benchmarking, and the Impact Analyser derives the total performance impact of each possible monitoring configuration. In step 3, the monitoring requirements and associated penalties for not meeting them are identified from analysing SLAs, policies and laws, and the Requirements Analyser transforms these requirements into a set of utility functions that define the benefit gained from monitoring a quality whilst a specific level of performance is being achieved. In step 4, the Optimiser uses the set of utility functions from the Requirements Analyser to score all monitoring configurations with impacts from the Impact Analyser, and the monitoring configuration that yields the highest utility is selected and applied to the monitoring system.
The Enumerator, Impact Analyser, Requirements Analyser, and Optimiser components are described below. Section 8.2 contains details of the implementation of each of these components.

5.3.1 Enumerator

The Enumerator takes as input the set of available monitors, and outputs the set of all possible monitoring configurations.

A monitor can monitor one or more quality types of one or more services, simultaneously or at different times. For a particular system, we refer to the set of all quality types as \( Q \). For iTravel, the set \( Q \) is \{response time, security, privacy, reliability\}. We refer to the set of all services as \( S \). For iTravel, the set \( S \) is \{book_flight, flight_info\}. We refer to the set of all monitors as \( M \). For iTravel, the set \( M \) is \{Generic Eavesdropper, book_flight Interceptor, book_flight Probe, flight_info Interceptor, flight_info Probe\}.

We refer to the ability of a monitor \( m \in M \) to monitor a quality type \( q \in Q \) of a service \( s \in S \) as a monitoring capability, \( mcap = (m, s, q) \). For example, one of iTravel’s monitoring capabilities is \((\text{Generic Eavesdropper}, \text{response time}, \text{book_flight})\), representing the ability of the \text{Generic Eavesdropper} to measure the response time of the \text{book_flight} service.

Each monitoring capability in the system has a set of possible monitoring levels \( ML \). These levels are integers that represent a level of monitoring. This level of monitoring may be resolution, sampling rate, or some other more complex aspect such as whether to parse just a message header, or a message header and body. Each monitoring capability has at least two monitoring levels (representing no monitoring and complete or highest-level monitoring).

When a monitoring capability \((m, s, q)\) is set with a particular monitoring level \( ml \in ML \), it is called a monitoring setting, \( ms = ((m, s, q) \mapsto ml) \). For example, \(((\text{Generic Eavesdropper}, \text{response time}, \text{book_flight}) \mapsto 1)\) represents the \text{Generic Eavesdropper} monitor observing response time of the \text{book_flight} service at moni-
monitoring levels of 0, 0.5, or 1.0, and monitor capabilities in the system. The particular monitoring settings obtained by assigning each of the monitoring capabilities in a system with a monitoring level of its own is called a monitoring configuration, $mc \in MC$ and $mc = \{(m,s,q) \mapsto ml : ML \forall (m,s,q) \in MCAP\}$, where $MCAP$ is the set of all available monitoring capabilities in the system.

For example, if we have monitor $m_1$ capable of monitoring quality $q_1$ of service $s_1$ at levels of 0, 0.5, or 1.0, and monitor $m_2$, capable of simultaneously monitoring quality $q_1$ and $q_2$ of service $s_1$ at levels of 0 or 1.0, the three monitoring capabilities are: \{(m_1, s_1, q_1), (m_2, s_1, q_1), (m_2, s_1, q_2)\}, and one of the possible monitoring configurations is: $mc = \{(m_1, s_1, q_1) \mapsto 0.5, (m_2, s_1, q_1) \mapsto 0, (m_2, s_1, q_2) \mapsto 1.0\}$.

The Enumerator generates all possible monitoring configurations $MC$ by stepping through all the allowable combinations of monitors, services, qualities and monitoring levels or, in other words, by assigning the different allowable permutations of monitoring levels to the set of all monitoring capabilities. For the above example, the output from the Enumerator (i.e., all its monitoring configurations) is presented in Table 5.1, where a row corresponds to a monitoring capability, a column corresponds to a monitoring configuration, and consequently a cell is a monitoring setting (the specific monitoring level of the corresponding monitoring capability in the relevant monitoring configuration). The particular monitoring configuration given above is column $mc_8$ in Table 5.1.
Not reflected in Table 5.1, the Enumerator actually further removes monitoring configurations that have duplicate monitoring, i.e., monitoring the same quality of the same service by more than one monitor, as we assume that there is no benefit from monitoring the same quality of the same service more than once. From Table 5.1, the overlapping monitoring configurations are mc5, mc6, mc11 and mc12, in which two monitors $m_1$ and $m_2$ are set to monitor the same quality $q_1$ of the same service $s_1$.

5.3.2 Impact Analyser

The Impact Analyser takes as input the set of all possible monitoring configurations $MC$ (e.g., Table 5.1), the set of all monitor setting impacts $MI$ and the set of all monitor overheads $MO$, and outputs the set of monitoring configurations with impacts $MCI$.

We use impact level $il$ to denote the monitoring impact on a quality $q$ of a service $s$, $(s,q) \mapsto il$, where the impact level $il \in IL$ is a percentage of the quality impact over the original quality (without the impact). For example, $((s_1,q_1) \mapsto 0.5)$ means the level of $q_1$ of $s_1$ will be half of what it was - if $q_1$ is response time, then response time would be doubled.

Monitoring has two kinds of performance impacts: monitoring overhead impacts, and monitor setting impacts.

The monitoring overhead impact $mo$ of a monitor $m$ on a relevant service $s$ occurs whenever the monitor is in use, regardless of the monitoring levels of the monitor, or what qualities or services it is monitoring. For example, an interceptor $m_1$ that redirects messages of services $s_1$ and $s_2$ for analysis may reduce response time of all messages passing through it by 10%, whether or not those messages are actually analysed. Let $SQ$ be all the service-quality pairs of concern in the system, the overheads a monitor has on every relevant quality $q$ of every service $s$ are $mo = \{(s,q) \mapsto il : il \in IL \forall (s,q) \in SQ\}$. Let $MIS = SQ \rightarrow IL$, the set of all monitor overheads is the collection of all the monitors’ overheads, $MO = \{m \mapsto mo : P(MIS) | \forall m \in M\}$. The overheads in the above example are a member of $MO$: $(m_1 \mapsto \{(s_1,q_1) \mapsto 0.1, (s_2,q_1) \mapsto 0.1\}) \in MO$, where $q_1$ represents response time.

The monitoring setting impacts $msi$ represent the impacts that a monitoring
setting $ms$ has on each quality $q$ of each service $s$ in the system. That is, $msi = \{(s, q) \mapsto il: IL(q, s) \in SQ\}$. For example, IT records may show that when monitor $m_1$ is configured to measure $q_1$ (response time) of service $s_1$ at monitoring level 0.5, it reduces $q_1$ of services $s_1$ and $s_2$ by 10%. The impacts of this monitor setting are $\{(s_1, q_1) \mapsto 0.1, (s_2, q_1) \mapsto 0.1\}$. The set of impacts of all possible monitor settings in $MS$ is: $MI = \{ms \mapsto msi : \mathcal{P}(MIS)|\forall ms \in MS\}$. For example, the above particular monitor setting’s impacts are a member of the overall $MI$: $\{(m_1, s_1, q_1) \mapsto 0.5\} \in MI$.

The impact of a monitoring configuration $mc$ on quality $q$ of service $s$ is made up the following components:

1. The sum of the impacts of all monitoring settings $ms \in mc$ on quality $q$ of service $s$: $I_{s,q}(mc) = \sum_{ms \in mc} MI(ms)(s, q)$; and

2. the sum of all the monitors’ overheads on quality $q$ of service $s$: $O_{s,q}(mc) = \sum_{m \in M} MO(m)(s, q)|\exists mc_i \in mc, mc_i \in mc, m = m, mc_i, ml > 0$.

Therefore, $mc$’s impacts on quality $q$ of service $s$ are: $IM(mc)(s, q) = I_{s,q}(mc) + O_{s,q}(mc)$ and $mc$’s impacts on all $(s, q)$ pairs are: $IM(mc) = \{(s, q) \mapsto IM(mc)(s, q)|\forall (s, q) \in SQ\}$. Finally, the set of all monitoring configurations with impacts (on service-quality pairs) is: $MCI = \{mc \mapsto IM(mc)|\forall mc \in MC\}$. The Impact Analyser calculates and outputs this set of monitoring configurations with impacts.

5.3.3 Requirements Analyser

The Requirements Analyser takes requirements from SLAs and other sources such as corporate governance as input, and outputs a set of utility functions representing the values of achieving these requirements.

We assume that requirements can be translated into statements that describe a monetary penalty such as a fine or refund for not meeting or not monitoring a quality measure for a service. For example, let us consider an SLA that requires service $s_1$ to have a response time of 15 seconds and a monitoring level (representing a sampling rate) of 50% to
demonstrate the QoS has been met, with a penalty of $100 for exceeding this. We say that the benefit is $100 for monitoring service $s_1$ for quality $q_1$ (response time) whilst keeping that response time less than 15 seconds with a monitoring level of at least 0.5.

Following from the above, a requirement that the quality $q$ of service $s$ must have a quality level of at least $ql$ and monitored at monitoring level $ml$ can be represented as $r = (s, q, ql, ml)$. The same requirement $r$ with a penalty for non-compliance can be represented as $rp = ((s, q, ql, ml) \mapsto \text{penalty})$. We model the quality level $ql$ as a fraction of the required quality level over the ideal quality level that is provided by the service. For example, if the best possible response time of a service is 10 seconds, and the required response time is 20 seconds, then the required quality level $ql$ is 0.5. The penalty is the monetary penalty that must be paid if the monitoring level $ml$ or the quality level $ql$ are not met. The set of all requirements with penalties is represented as $RP = \{(s, q, ql, ml) \mapsto \text{penalty} : \mathbb{R} \mid \forall (s, q, ql, ml) \in R\}$, where $R$ is the set of all requirements.

Consider the requirements that the quality $q$ of service $s$ should be monitored all the time and achieve a quality level of 0.75, with a penalty of $60 if the quality level goes below 0.75, and a penalty of $90 if the quality level goes below 0.5. This set of requirements with penalties can be represented as $rps = \{(s_1, q_1, 0.50, 1) \mapsto 90, (s_1, q_1, 0.75, 1) \mapsto 60, (s_1, q_1, 1, 1) \mapsto 0\}.$

We transform each monitoring requirement describing penalties for non-compliance into a set of utility functions describing benefit (utility) for compliance. This is based on the assumption that all value for monitoring comes from requirements. That is, we assume there is no value or benefit from good will for providing or monitoring a service beyond that required by SLAs. However, this good will may be explicitly added as requirements with appropriately sized values, if required.

Since our utilities are cumulative, for $rp = ((s, q, ql, ml) \mapsto \text{penalty})$, we have the corresponding utility:

$$\text{util}(s, q, ql, ml) = (\text{curmax}RP(s, q, ql, ml)) - \sum_{lr \in LR(s, q, sl, ml)}(\text{curmax} - \text{util}(lr)),$$

where $\text{curmax}$ is the benefit of achieving the quality level $ql$ and monitoring level $ml$, and $LR$ is the set of requirements with a quality level $lr.ql < ql$ or monitoring level $lr.ml < ml$, and
\[ LR(s, q, ql, ml) = \{(s, q, ql_1, ml_1) \mid \forall (s, q, ql_1, ml_1) \in R((ql_1 < q_1) \land (ml_1 \leq ml)) \lor ((ql_1 \leq q_1) \land (ml_1 < ml)) \}. \]

Therefore, the utility function covering all the monitoring requirements is:

\[ U = \{(s, q, ql, ml) \mapsto \text{util}(s, q, ql, ml) \mid (s, q, ql, ml) \in R \}. \]

For the above set of example requirements \( rps \), we have the following corresponding utilities:

\[ us = \{(s_1, q_1, .50, 1) \mapsto 0, (s_1, q_1, .75, 1) \mapsto 30, (s_1, q_1, 1, 1) \mapsto 60 \}. \]

So, the last element \(((s_1, q-1, 1, 1) \mapsto 60)\) of the utility set \( us \) is read as “achieving quality level greater than .75 of quality \( q_1 \) for service \( s_1 \) with a monitoring level 1 is worth an additional $60 (relative to the next lower quality level of 0.75), since fines of that amount will not have to be paid”. The absolute utility for the \( ql = 1 \) level is $90, including the utility at the lower quality level ($30).

Note that as in normal practice, once the requirements have been transformed to utilities the values of fines are normalised so that the highest fine \( f_h \) becomes 1.0, and all other fines \( f_i \) become fractions \( \frac{f_i}{f_h} \). Also note that the translation from requirements into the requirements representation \( R \) (and then to the utility function) is performed manually. Furthermore, designers may add custom utilities that are not directly derived from requirements.

### 5.3.4 Optimiser

The Optimiser takes as input the set of utilities \( U \), the set of monitoring configurations with impacts \( MCI \), and the ideal service quality levels for all qualities of all services \( IQL = \{(s, q) \mapsto iql : QL \mid \forall s \in S, \forall q \in Q\}, \) where \( iql \) is the pre-determined best achievable quality level for quality \( q \) of service \( s \). The Optimiser outputs a monitoring configuration that gives the highest utility in terms of meeting requirements and reducing monitoring impacts.

For a monitoring configuration \( mc \), the Optimiser first obtains the Final Quality Levels \( FQL \) of all \((s, q)\) pairs by subtracting all relevant impacts in \( MCI \) from each \((s, q)\) pair’s ideal quality level \( IQL \), i.e.:

\[ FQL(mc) = \{(s, q) \mapsto fql : QL \mid iql = IQL(s, q) - MCI(mc)(s, q)\}. \]
The utility concerning the \((s,q)\) pair under monitoring configuration \(mc\), \(u(mc)(s,q)\), is the sum of those utilities concerning \((s,q)\):

1. Whose required quality level \(ql\) has been met by the pair’s final quality level \(FQL(mc)(s,q)\), and;
2. Whose monitoring level \(ml\) is met by at least one monitor setting in \(mc\).

Let \(R_{(s,q)}\) be the set of all requirements concerning \((s,q)\), i.e.,
\[
R_{(s,q)} = \{(s_1,q_1,ql,ml) | (s_1,q_1,ql,ml) \in R \land s_1 = s \land q_1 = q\}.
\]
Then,
\[
u(mc)(s,q) = \sum_{(s,q,ql,ml) \in R_{(s,q)}} (\text{util}(m,s,ql,ml) \times a \times b),
\]
where
\[
a = 1 \text{ if } FQL(s,q) \geq ql, \text{ or } a = 0 \text{ otherwise};
\]
\[
b = 1 \text{ if } (\exists m \in M, ms(m,s,q) \geq ml), \text{ or } b = 0 \text{ otherwise}.
\]
Note that the where clause indicates the two conditions (stated above) for the relevant utility to be included. Condition \(a\) indicates that the required quality level was met, and condition \(b\) indicates that the required monitoring level for that QoS was used.

The total utility for the monitoring configuration \(mc\) will be the sum of the utilities for all \((s,q)\) pairs under the configuration, \(u(mc)\). Let
\[
SQ_{mc} = \{(s,q) | \forall (m,s,q,ml) \in mc\}.
\]
Then,
\[
u(mc) = \sum_{(s,q) \in SQ_{mc}} u(mc)(s,q).
\]
The set of all monitoring configurations with utility is:
\[
MCU = \{mc \mapsto u(mc) | mc \in MC\}.
\]
A monitoring configuration that gives the highest total utility will be an optimal monitoring configuration, \(optimal_{mc} \in \{mc_1 \forall mc_2 \in MC, MCU(mc_1) \geq MCU(mc_2)\}\), which may be applied to the monitors in the system either manually or automatically (if available via methods such as Simple Network Management Protocol (SNMP) or WS-Management). It should be noted that our implementation did not use or define a new language for monitoring configuration. Instead, we used a simple syntax defining a monitoring setting as a comma-separated string containing a monitor ID, a service ID, a quality type ID, and a monitoring level that could be interpreted by the monitor.

**Optimisation Complexity** As demonstrated in Table 5.1, the number of monitoring configurations is the product of the number of monitoring levels \((ML_{mcap})\) for all monitoring capabilities in the system \((mcap \in MCAP)\), as defined in Equation 5.1. For Table
5.1 The number of monitoring configurations is the number of columns \((3 \times 2 \times 2 = 12)\), which is reduced to 8 once the monitoring configurations with service quality coverage overlap \((mc_5, mc_6, mc_{11} \text{ and } mc_{12})\) are pruned out.

\[
\prod_{mc_{\text{cap}} \in MCAP} |ML_{mc_{\text{cap}}}| \tag{5.1}
\]

Therefore, the size of this search space is bounded by:

\[
\mathcal{O}(\text{avg}(|ML_{mc_{\text{cap}}}|)^{|MCAP|}) \tag{5.2}
\]

Since we step through the search space once for each of Impact Analysis and Optimisation, this is also the time complexity for our optimisation technique.

As such, the problem of searching for an optimal monitoring configuration is a combinatorial optimisation problem with both overhead (fixed) and instance (variable) costs, analogous to an integer Fixed Charge Network Flow Problem (FCNFP), demonstrated to be NP-hard [63]. In this instance, we have provided an enumerative (brute-force) technique with the expected exponential time bound described above. This is for three reasons. First, one purpose of this research is to demonstrate the possible benefits of applying optimisation in terms of performance and utility increases, and the computation time for the scenario was not seen as of prime concern for this purpose. Second, we wish to have a baseline of a ‘perfect’ optimisation (i.e., obtaining a true optimal monitoring configuration) against which to compare future heuristic algorithms for optimal or sub-optimal solutions, such as the heuristic described in Chapter 6. Third, the technique is still usable for a system with a small deployment of monitors but still too complex for a human to efficiently solve. For example, the technique can be used for cases of up to 15 monitoring capabilities with 3 monitoring levels each, i.e., a search space in the order of \(10^7\).

5.4 iTravel Optimisation

In order to measure the effectiveness of the optimisation framework and associated techniques described above, we have applied the optimisation framework to the iTravel scenario. Below, we introduce the prototype implementation of the framework and its application to the iTravel scenario.
For the optimisation framework, we have implemented the Enumerator, Impact Analyser, and Optimiser components as a set of platform-independent Perl scripts. As previously stated, the Requirements Analyser has not been automated, so this function was performed manually.

5.4.1 iTravel Implementation

We have prototyped the iTravel scenario (described in detail in Chapter 2), as depicted in Figure 5.3. The system has been implemented using Apache Tomcat/Axis on Ubuntu 9.04. Full details of the implementation are provided in Section 8.2.

5.4.2 iTravel System Performance

The response time for both the flight_info and book_flight services in the iTravel scenario has been measured to be 2.5 seconds under ideal conditions of no other system load and no monitoring. All other aspects (security, reliability, and privacy) are assumed 100% for our tests (i.e., we assume that the security, reliability, and privacy properties of services will always be met).

The set of ideal quality levels iIQL shows the ideal response time of both services to be 2.5 seconds, and each other property (security, privacy, and reliability) to be 100%:

\[ \text{IQL} = \{ \]
\[ (\text{flight}_\text{info}, \text{responsetime}) \mapsto 2.5s; \]
\[ (\text{book}_\text{flight}, \text{responsetime}) \mapsto 2.5s; \]
(flight_info, Reliability) $\mapsto$ 100%;
(book_flight, Reliability) $\mapsto$ 100%;
(flight_info, privacy) $\mapsto$ 100%;
(book_flight, privacy) $\mapsto$ 100%;
(flight_info, security) $\mapsto$ 100%;
(book_flight, security) $\mapsto$ 100%
}

5.4.3 iTravel Monitoring Settings

The enumeration input for iTravel is the set of monitoring settings and levels, $iMS$, where:

\[
\begin{align*}
\text{iMS} &= \{ \\
(flight\_infoProbe, flight\_info, security) &\mapsto \{0, 1\}, \\
(flight\_infoProbe, flight\_info, response\_time) &\mapsto \{0, 1\}, \\
(flight\_infoProbe, flight\_info, Reliability) &\mapsto \{0, 1\}, \\
(flight\_infoProbe, flight\_info, privacy) &\mapsto \{0, 1\}, \\
\ldots
\end{align*}
\]

... 

(GenericEavesdropper, flight_info, response\_time) $\mapsto$ \{0, 1\},

(GenericEavesdropper, book_flight, response\_time) $\mapsto$ \{0, 1\}

).

The output $iMC$ of the Enumerator for iTravel is then the total enumeration of the input $iMS$ set, as shown in Table 5.2 (for clarity Table 5.2 does not have duplicated monitoring configurations pruned out). In Table 5.2, each monitoring capability is one row (e.g. the first row is monitoring capability $m_1, s_1, q_1$, representing monitor $m_1$ observing quality type $q_1$ of service $s_1$; each monitoring setting is one cell (e.g. the top left cell represents monitoring capability $m_1, s_1, q_1$ being assigned a monitoring level of 0); and each monitoring configuration is one column (e.g. column $mc_1$ represents a monitoring configuration where all monitors are disabled).
Table 5.2: iTravel Monitoring Enumeration

<table>
<thead>
<tr>
<th></th>
<th>$m_{c_1}$</th>
<th>$m_{c_2}$</th>
<th>$m_{c_3}$</th>
<th>$m_{c_4}$</th>
<th>$m_{c_5}$</th>
<th>\ldots</th>
<th>$m_{262145}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_{1, s_1, q_1}$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>\ldots</td>
<td>1</td>
</tr>
<tr>
<td>$m_{1, s_1, q_2}$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>\ldots</td>
<td>1</td>
</tr>
<tr>
<td>$m_{1, s_1, q_3}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>\ldots</td>
<td>1</td>
</tr>
<tr>
<td>$m_{1, s_1, q_4}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>1</td>
</tr>
<tr>
<td>$m_{2, s_2, q_1}$</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>$m_{5, s_1, q_1}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>1</td>
</tr>
<tr>
<td>$m_{5, s_2, q_1}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>1</td>
</tr>
</tbody>
</table>
5.4.4 iTravel Requirements Analysis

Recall from Chapter 2 that iTravel is bound by a set of SLA and corporate governance requirements:

SLA1. flight_info response time will be under 10s, penalty=33% account fee

SLA2. book_flight response time will be under 10s, penalty=66% account fee

SLA3. flight_info will be at least 95% reliable, penalty=100% account fee

SLA4. book_flight will be at least 98% reliable, penalty=100% account fee

SLA5. flight_info will meet privacy requirements, penalty=100% account fee

SLA6. book_flight will meet privacy requirements, penalty=100% account fee

CG1 flight_info meets security requirements, penalty=$10,000

CG2. book_flight meets security requirements, penalty=$10,000

CG3. flight_info meets privacy requirements, penalty=$5,000

CG4. book_flight meets privacy requirements, penalty=$5,000

For example, SLA1 states that iTravel must deliver a response time of under 10 seconds for the flight_info service, otherwise the consumer must be refunded 33% of their monthly account fee (which is $100).

The Requirements Analyser takes as input the iTravel requirements iR (summarised below), and outputs a set of utility functions iU describing the utility to be gained for meeting these requirements. In our case, utility is directly and linearly computed from fines for non-compliance, however the set of utility functions may be modified or extended to add custom requirements or model some non-linear relationship.

The set of iTravel requirements iR is:

{  
  (flight_info, responsetime, 0.25, 1) \mapsto 0.33x,  
  (book_flight, responsetime, 0.25, 1) \mapsto 0.66x,  
  (flight_info, Reliability, 0.95, 1) \mapsto 1.0x,  
}
(book_flight, Reliability, 0.98, 1) $\mapsto$ 1.0x,
(flight_info, privacy, 1, 1) $\mapsto$ 1.0x,
(book_flight, privacy, 1, 1) $\mapsto$ 1.0x,
(flight_info, privacy, 1, 1) $\mapsto$ $10,000,
(book_flight, privacy, 1, 1) $\mapsto$ $10,000,
(flight_info, security, 1, 1) $\mapsto$ $5,000,
(book_flight, security, 1, 1) $\mapsto$ $5,000
}

Where $x$ represents the monthly service fee ($100) and current number of consumers (100), i.e. $0.33x = 0.33 \times 100 \times 100 = 3300$. Note that the quality levels for the first two requirements are 0.25 - this is because the initial iTravel requirements gave 10 seconds as the minimum acceptable response time, and the current ideal response time for each service is 2.5 seconds. The iTravel requirements $iR$ are transformed into the set of iTravel utility functions $iU$:

{  
(flight_info, responsetime, 0.25, 1) $\mapsto$ 0.165(3300),
(book_flight, responsetime, 0.25, 1) $\mapsto$ 0.33(6600),
(flight_info, Reliability, 1, 1) $\mapsto$ 0.5(10,000),
(book_flight, Reliability, 1, 1) $\mapsto$ 0.5(10,000),
(flight_info, privacy, 1, 1) $\mapsto$ 1(20,000),
(book_flight, privacy, 1, 1) $\mapsto$ 1(20,000),
(flight_info, security, 1, 1) $\mapsto$ 0.25(5,000),
(book_flight, security, 1, 1) $\mapsto$ 0.5(10,000)
}

The set $iU$ describes the utility to be gained from meeting each monitoring requirement. Utilities have been normalised to values between 0 and 1. Actual monetary values are shown in brackets, e.g. 1 (20,000) to demonstrate the derivation and normalisation of the utility levels.
5.4.5 iTravel Monitoring Impact Analysis

We have measured the response time impacts that occur from monitoring in the iTravel system, and present the results below. These impacts are an average percentage increase of response time, compared with the response time of a service without monitoring\(^4\). For example, using the book_flight Interceptor at 100% will increase response time by 2% for the book_flight service, plus 1% for each property measured (security, response time, privacy, and reliability). Additionally, measuring one or more qualities with the book_flight Interceptor impacts the response time of the flight_info service by 1%. The eavesdropper and probe monitors have a fixed impact regardless of how many qualities of service are monitored, e.g. the impact of monitoring security, response time, reliability and privacy of flight_info using the flight_info Probe is the same as the impact of monitoring only security. Note than some monitors have an impact on a service that they aren’t monitoring. For example, the flight_info Probe has an overhead impact that reduces the response time of book_flight. These impacts occur due to monitors and services sharing the same, limited set of resources.

The Impact Analyser takes as input the monitoring configuration (Table 5.2), iTravel monitoring setting impacts iMSI, and iTravel monitoring overheads iMO. For iTravel, the set of monitoring setting impacts iMSI (derived from benchmarks) is:

\[
\begin{align*}
&\{ (flight\_infoProbe, \text{responsetime}, flight\_info, ml_1) \rightarrow \emptyset \}; \\
&\{ (flight\_infoProbe, \text{Reliability}, flight\_info, ml_1) \rightarrow \emptyset \}; \\
&\{ (flight\_infoProbe, \text{privacy}, flight\_info, ml_1) \rightarrow \emptyset \}; \\
&\{ (flight\_infoProbe, \text{security}, flight\_info, ml_1) \rightarrow \emptyset \}; \\
&\{ (book\_flightProbe, \text{responsetime}, book\_flight, ml_1) \rightarrow \emptyset \}; \\
&\{ (book\_flightProbe, \text{Reliability}, book\_flight, ml_1) \rightarrow \emptyset \}; \\
&\{ (book\_flightProbe, \text{privacy}, book\_flight, ml_1) \rightarrow \emptyset \}; \\
&\{ (book\_flightProbe, \text{security}, book\_flight, ml_1) \rightarrow \emptyset \}; \\
&\{ (flight\_infoInterceptor, \text{responsetime}, flight\_info, ml_1) \rightarrow \\
&\quad \{ (flight\_info, \text{responsetime}, 0.01), (book\_flight, \text{responsetime}, 0.005) \}\};
\end{align*}
\]

\(^4\)Whilst we have focussed on response time, any other measurable QoS may be substituted. For example, if monitors in the system measure and impact throughput or reliability, then these property may be analysed in the same way.
\{(\text{flight\_infoInterceptor}, \text{Reliability}, \text{flight\_info}, ml_1) \rightarrow \\
\{(\text{flight\_info}, \text{responsetime}, 0.01), (\text{book\_flight}, \text{responsetime}, 0.005))\}; \\
\{(\text{flight\_infoInterceptor}, \text{privacy}, \text{flight\_info}, ml_1) \rightarrow \\
\{(\text{flight\_info}, \text{responsetime}, 0.01), (\text{book\_flight}, \text{responsetime}, 0.005))\}; \\
\{(\text{flight\_infoInterceptor}, \text{security}, \text{flight\_info}, ml_1) \rightarrow \\
\{(\text{book\_flight}, \text{responsetime}, 0.01), (\text{flight\_info}, \text{responsetime}, 0.005))\}; \\
\{(\text{book\_flightInterceptor}, \text{responsetime}, \text{book\_flight}, ml_1) \rightarrow \\
\{(\text{book\_flight}, \text{responsetime}, 0.01), (\text{flight\_info}, \text{responsetime}, 0.005))\}; \\
\{(\text{book\_flightInterceptor}, \text{Reliability}, \text{book\_flight}, ml_1) \rightarrow \\
\{(\text{book\_flight}, \text{responsetime}, 0.01), (\text{flight\_info}, \text{responsetime}, 0.005))\}; \\
\{(\text{book\_flightInterceptor}, \text{privacy}, \text{book\_flight}, ml_1) \rightarrow \\
\{(\text{book\_flight}, \text{responsetime}, 0.01), (\text{flight\_info}, \text{responsetime}, 0.005))\}; \\
\{(\text{book\_flightInterceptor}, \text{security}, \text{book\_flight}, ml_1) \rightarrow \\
\{(\text{book\_flight}, \text{responsetime}, 0.01), (\text{flight\_info}, \text{responsetime}, 0.005))\}; \\
\{(\text{GenericEavesdropper}, \text{responsetime}, \text{flight\_info}, ml_1) \rightarrow \\
\{(\text{flight\_info}, \text{responsetime}, 0.01), (\text{flight\_info}, \text{responsetime}, 0.01))\}; \\
\{(\text{GenericEavesdropper}, \text{responsetime}, \text{book\_flight}, ml_1) \rightarrow \\
\{(\text{book\_flight}, \text{responsetime}, 0.01), (\text{flight\_info}, \text{responsetime}, 0.01))\}; \}

And the set of iTravel monitoring overheads iMO is

\{
(\text{flight\_infoProbe} \rightarrow \\
\{(\text{flight\_info}, \text{responsetime}, 0.08), (\text{book\_flight}, \text{responsetime}, 0.08))\}; \\
(\text{book\_flightProbe} \rightarrow \\
\{(\text{flight\_info}, \text{responsetime}, 0.08), (\text{book\_flight}, \text{responsetime}, 0.08))\}; \\
(\text{flight\_infoInterceptor} \rightarrow \\
\{(\text{flight\_info}, \text{responsetime}, 0.02))\}; \\
(\text{book\_flightInterceptor} \rightarrow \\
\{(\text{book\_flight}, \text{responsetime}, 0.02))\}; \\
(\text{GenericEavesdropper} \rightarrow \\
\{\})
\}
These sets $iMSI$, $iMO$ and $iMC$ are used by the Impact Analyser to calculate the set of monitoring configurations with impacts for iTravel, $iMCI$. Too large to list here, $iMCI$ maps each $iMC$ in Table 5.2 to a total set of monitoring impacts. For example, $iMC_{262145}$ has every monitor enabled, so the impact on response time is the total impact of every $iMSI$ and $iMO$, \(0.01 \times 4 + 0.005 \times 4 + 0.01 + 0.01\) + \(0.08 + 0.08 + 0.08 + 0.08 + 0.02\) = 0.42, for each service. This information on monitoring requirements and impacts, along with information recorded about the baseline response times for the iTravel Web Services has been used to perform optimisation of iTravel’s Web Service monitoring system.

### 5.4.6 iTravel Optimisation

The default ‘max’ monitoring configuration in which all monitors are enabled incurs the complete set of monitoring impacts in the iTravel scenario. As discussed above, the total impact on response time of each service under this scenario is 42%. This yields response times of \(2.5/(1 - 0.42) \approx 4.3\) seconds. This is the benchmark to which we compare our optimised solution.

The Optimiser takes as input the iTravel utility functions $iU$ (defined in Section 5.4.4), ideal quality levels $iIQL$ (defined in Section 5.4.2), and monitoring configurations with impacts $iMCI$ (defined in Section 5.4.5).

The configuration in $iMCI$ that yields the highest utility is selected and applied to the system. For iTravel, this configuration was:

```plaintext
{(flight_infoProbe,flight_info,responsetime..security) = 0,
(book_flightProbe,book_flight,responsetime..security) = 0,
(flight_infoInterceptor,flight_info,responsetime..4) = 1,
(book_flightInterceptor,Reliability,responsetime..4) = 1,
(GenericEavesdropper,flight_info..2,responsetime) = 0}
```

i.e., the flight_info Interceptor and book_flight Interceptor were both set to monitor every quality of their respective target services, and every other monitor was disabled. The impact of this configuration is the sum of the impacts for each service in $iMI$ (0.01 times 4 + 0.005 times 4) and $iMO$ (0.02). Therefore, the total impacts were 8% per service, and
the resulting response times of each service were \(2.5/(1 - 0.08) \approx 2.7\), under ideal load. This optimal configuration meets every requirement, and so receives the maximum business utility of 4.245, which corresponds to $84,900 in business value. It should be noted that the default max configuration also receives this utility level, since the response time under the max configuration is 4.3 seconds and the maximum response time requirement is 10 seconds. However, the optimised solution provides a lower response time, and so is more likely to meet the response time goals as load on the system increases. For this reason, the Optimiser always selects a monitoring configuration with the lowest average response time from the set of optimal monitoring configurations.

5.5 Generation of Optimisation Problems

The previous sections applied our approach to a single scenario (iTravel) in order to demonstrate our optimisation technique and its feasibility. In order to demonstrate that our approach will apply to a wide range of application scenarios, and to quantify the efficiency and accuracy of the approach, we have developed a set of tools for generating randomised problem input sets for optimisation. These randomly generated input sets consist of sets of monitors, services, quality types, monitoring impacts, baseline qualities and utilities (representing requirements on qualities of service). The input sets are generated both purely randomly, and randomly with trends. Purely random input sets remove any bias in generating the input system structures, whilst the trended random input sets allow for more realistic systems to be tested.

The Pure Random Generators take upper and lower bounds and generate random input sets within these. For example, sets of monitors may be generated, with each problem set having between 1 and 10 monitors as upper and lower bounds. Extending this, the Trended Random Generators attempt to generate random input sets that follow realistic trends. For example, monitoring a service will affect that service more than any other service - this is one trend. These trends are modelled in the Trended Random Generators, so that the output problem sets are realistic, and the corresponding experiments that use those problem sets are more representative of real-world systems.

The rest of this section describes the purely random and trended random generators developed in greater detail.
5.5.1 Monitoring Capability Generators

The Monitoring Capability Generators take a set of control parameters as input and output a set of monitoring capabilities, which define the monitors in the system, and what services, quality types, and monitoring levels they can monitor.

The input control parameters of the Monitoring Capability Generators are the maximum number of services, maximum number of quality types, maximum number of monitoring levels, likelihood of a service being monitored, likelihood of a quality type being monitored, and likelihood of a monitoring level being used. For example, we could run the generator with “5 services, 5 quality types, 10 monitoring levels, 10% probability that a monitor is selected to observe a service, 50% probability that a quality type is monitored, and 80% probability that a monitoring level is selected”. In this case, for each of the 5 services, there is a 10% chance a matching monitor will be selected, followed by a 50% chance of one of the five quality types being selected from that monitor, followed by an 80% chance of selecting each of the 10 monitoring levels. This would give $10\% \times 50\% \times 80\% = 0.04\%$ probability that any service/quality type/monitoring level will be output as a possible monitoring setting.

The output of each of the Monitoring Capability Generators is a Monitoring Capabilities File, with columns of service, monitor, quality type, and monitoring level, and each row representing a capability of a monitor to observe a Service’s quality of type quality type at a monitoring level. For example, one line of the Monitoring Capabilities File may be [book_flight, book_flight interceptor, response time, 0.1].

5.5.1.1 Pure Random Monitoring Capability Generator

The Pure Random Monitoring Capability Generator takes the inputs described above (the maximum number of services, maximum number of quality types, maximum number of monitoring levels, likelihood of a service being monitored, likelihood of a quality type being monitored, and likelihood of a monitoring level being used), and outputs a Monitoring Configuration File. The data in this Monitoring Configuration File is purely randomly generated, based on the supplied input parameters. The Pure Random Monitoring Capability Generator achieves this by performing the actions described in Algorithm 1.
Algorithm 1 Pure Random Monitoring Capability Generator

1: for all s in Services[] do
2:   for all m in Monitors[] do
3:     if rand(1) ≤ pr_service_monitored then
4:       for all q in Quality_Types[] do
5:         if rand(1) ≤ pr_quality_monitored then
6:           for all ml in Monitoring_Levels[] do
7:             if rand(1) ≤ pr_monitoring_level_used then
8:               append(Monitoring_Capabilities[], (s, m, q, ml) )
9:             end if
10:           end for
11:         end if
12:       end for
13:     end if
14:   end for
15: end for
16: print Monitoring_Capabilities[]
In Algorithm 1, a set of monitoring capabilities is generated (lines 1-15) and printed (line 16). The Monitoring Configuration File is generated by enumerating through each service (line 1), then each monitor (line 2), then each quality type (line 4), and finally each monitoring level (line 6), and outputting a monitoring capability for that (service/monitor/quality type/monitoring level) if random cutoff criteria are met. If all of the likelihoods were given as 1, then the output would be services × monitors × Quality-Types × Monitoring-Levels (the Cartesian product of the sets of all services, monitors, quality types and monitoring levels). Otherwise, depending on the likelihoods given, some monitors may be skipped (line 3), some quality types may be skipped (line 5), and some monitoring levels may be skipped (line 7).

5.5.1.2 Trended Random Monitoring Capability Generator

The Trended Random Monitoring Capability Generator extends the Pure Random Monitoring Capability Generator by increasing the probability of:

(1) A monitor observing the same quality type on additional services

(2) A monitor observing additional quality types on the same service

(3) A monitor observing a (quality, service) pair with additional monitoring levels

Rule 1 is based on the assumption that monitors are more likely to be able to observe the same quality type of multiple services than a purely random distribution of quality types. For example, the iTravel Generic Eavesdropper monitor observes response time on both services, and no other quality types. This is considered more likely than a monitor that observes response time of one service, and security of another service.

Rule 2 is based on the assumption that a monitor is more likely to be able to observe the same service for multiple quality types than a purely random distribution of services. For example, the iTravel flight_info Probe monitor observes the flight_info service for response time, privacy, security and Reliability, but does not observe any other service.

Rule 3 is based on the assumption that a monitor is more likely to be able to observe the same quality type of the same service at multiple monitoring levels than a purely random distribution of quality types and services. This reflects that if a monitor can
observe, say, response time of the flight_info service at a monitoring level of 10%, then that monitor now has a much higher probability of being able to observe response time of the flight_info service at a monitoring level of 20% than if it could not monitor response time of the flight_info service at a monitoring level of 10%.

Based on these new rules, the Trended Random Monitoring Capability Generator performs the actions described in Algorithm 2.

Algorithm 2 Trended Random Monitoring Capability Generator

1: for all s in Services[] do
2:   for all m in Monitors[] do
3:     if ( (exists(Monitoring_Capabilities[(*,s,m,*)] and (rand(1) \leq pr_service_duplicated)) or (exists(Monitoring_Capabilities[(*,s,m,*)] and (rand(1) \leq pr_service_monitored)))) then
4:       for all q in Quality_Types[] do
5:         if ( (exists(Monitoring_Capabilities[(*,m,q,*)] and (rand(1) \leq pr_quality_monitored_duplicated)) or (exists(Monitoring_Capabilities[(*,m,q,*)] and (rand(1) \leq pr_quality_monitored))) then
6:           for all ml in Monitoring_Levels[] do
7:             if ( (exists(Monitoring_Capabilities[(s,m,q,*)] and (rand(1) \leq pr_monitoring_level_used_duplicated)) or (exists(Monitoring_Capabilities[(s,m,q,*)] and (rand(1) \leq pr_monitoring_level_used))) then
8:               append(Monitoring_Capabilities[], (s, m, q, ml) )
9:             end if
10:           end for
11:         end if
12:       end for
13:     end if
14:   end for
15: end for
16: print Monitoring_Capabilities[]

As stated, the Trended Random Monitoring Capability Generator described in Algorithm 2 is the same as the Pure Random Monitoring Capability Generator
described in Algorithm 1, with exceptions for invoking rules 1-3 defined earlier. Line 3 invokes Rule 1, line 5 invokes Rule 2, and line 7 invokes Rule 3. Line 3 invokes Rule 1 by first checking if the current monitor already observes the current service, and if so uses the pr_service_duplicated likelihood for that monitor observing that service again. Conversely, if the current monitor does not already observe the current service, then the default pr_service_monitored likelihood is used. Line 5 performs the same actions on quality types, using the pr_quality_monitored_duplicated likelihood, and Line 7 performs the same actions on monitoring levels using the pr_monitoring_level_used_duplicated likelihood.

The duplicated probabilities pr_service_duplicated, pr_quality_monitored_duplicated, and pr_monitoring_level_duplicated can be taken as input parameters. However, experimentation values of 0.6 for pr_service_duplicated, 0.9 for pr_quality_duplicated, and 0.95 for pr_monitoring_level_duplicated consistently yield the most realistic Monitoring Configurations (that is, configurations with sets of monitors and services that more closely resemble real systems from business or literature due to their set of monitoring capabilities).

5.5.2 Monitoring Impact Generators

The Monitoring Impact Generators take as input a Monitoring Capabilities File (generated by either the Pure Random Monitoring Capability Generator or the Trended Random Monitoring Capability Generator), an Impact Divisor, and quality type, and Service Impact Likelihoods. For each row in the Monitoring Capabilities File (representing a monitoring capability), the Monitoring Impact Generators output 0 or more randomly generated Monitoring Setting Impact (per-instance) costs associated with that monitoring capability. Additionally, for each monitor found in the Monitoring Capabilities File, the Monitoring Impact Generators output 0 or more Overhead Impacts. The Impact Divisor allows for the production of impact sets with overall higher and lower impacts. Each Monitoring Setting Impact and Overhead Impact is divided by the Impact Divisor, so that an Impact Divisor value of 0.5 will double the average impacts, and a value of 2 will halve the average impacts.
The output of the Monitoring Impact Generators is a Monitoring Impacts File, with columns of 
[ Service Monitored/ Quality Type Monitored/monitor/monitoring level→Quality Type Impacted/Service Impacted/Impact Level ] 
for Monitoring Setting Impacts, where one row represents one Monitoring Setting Impact. The file also contains of a set of Overhead Impacts, with
the format [Monitor→Quality Type Impacted/Service Impacted/Impact Level ],
where one row represents one Overhead Impact for a monitor.

5.5.2.1 Pure Random Impacts Generator

This generator builds a model of all the Monitoring Settings from the Monitoring Capabilities File to ensure that for each (service, monitor, quality type, monitoring level) → Monitoring Setting Impact, that Monitoring Setting Impact is not lower than the impacts of other monitoring settings with lower monitoring levels, i.e. that the Monitoring Setting Impact MSI for monitoring a service S and quality type Q using a monitor M at a monitoring level ML is never lower than the Monitoring Setting Impact of monitoring the same S and Q using the same M, and a monitoring level MLL < ML. Using this model of Monitoring Settings the generator outputs a Monitoring Setting Impact on a quality type and service for each monitor and monitoring capability, if a random cut-off is met.

Algorithm 5.5.1 Pure Random Impacts Generator

1: for all (m,s,q,ml) in Monitoring_Capabilities_File do
2: append( Monitoring_Capabilities{m,s,q}, ml )
3: end for
4: for all MC in Monitoring_Capabilities{} do
5: for all q in Quality_Types[] do
6: if rand(1) ≤ quality_cutoff then
7: for all s in Services[] do
8: if rand(1) ≤ service_cutoff then
9: impact_level ← 0
10: for all ml in Monitoring_Capabilities{MC.m,s,q} do
11: if ml == 0 then
12: impact_level ← 0
13: end if
14: append(Impacts[], (MC.s, MC.q, MC.m, ml, q, s, impact_level))
15: impact_level ← impact_level + (rand(1) / divisor)
16: end for
17: end if
18: end for
19: end if
20: end for
21: end for
22: for all m in Monitors[] do
23: for all q in Quality_Types[] do
24: if rand(1) ≤ quality_cutoff then
25: for all s in Services[] do
26: if rand(1) ≤ service_cutoff then
27: impact_level ← rand(1) / divsor
28: append(Overheads[], (m,s,q,impact_level))
29: end if
30: end for
31: end if
32: end for
33: end for
34: print Impacts[], Overheads[]

The Pure Random Monitoring Setting Impacts Generator is described in Algorithm 5.5.1. Lines 1-2 build a MonitoringCapabilities hash, lines 4-21 randomly generate a set of monitoring setting impacts (Impacts) for each monitoring capability, and lines 22-33 randomly generate a set of overhead impacts for each monitor.

The per-instance impacts are generated by enumerating each Monitoring Capability (Line 4), and then generating impacts for that monitoring capability under likelihoods for impacts on quality types (Line 6) and services (Line 8). If these cutoffs are met, then a
set of impacts are generated for that service and quality type (Lines 9-17). The Impact Level increases along with the monitoring level (Line 15).

The Overhead Impacts are generated by enumerating each monitor (Line 22), quality type (Line 23), and service (Line 25). If the Quality Type cutoff (Line 24) and the Service cutoff (Line 26) are met for a monitor, then an Overhead Impact is generated (Line 27) and added to the set of Overhead Impacts (Line 28).

5.5.2.2 Trended Random Impacts Generator

The Trended Random Monitoring Setting Impacts Generator extends the Pure Random Monitoring Setting Impacts Generator by randomly generating sets of impact and usage weights for all quality types, monitors, and services, and increasing the impact of monitoring based on the ‘closeness’ of services.

The set of impact weights is used so that each unique quality type, monitor, or service has similar impact levels associated with it. This is based on the assumptions that:

1. The impacts of monitoring a quality type will be similar, regardless of the monitor or service monitored. For example, the impact monitoring response time will usually be lower than the impact of monitoring for security, and this is independent of the monitor used or service monitored. In this example, the security monitor may be decrypting and parsing a message, and comparing the contents against a blacklist, whilst the response time monitor may just be logging ingoing and outgoing time stamps with unique message IDs and comparing their differences.

2. The impacts of monitoring a service will be similar, regardless of the monitor or quality type. For example, the impact of monitoring flight info will usually be lower than the impact of monitoring book_flight, and this is independent of the quality type monitored or monitor being used. In this example, the messages for book_flight may require more intrusive or intensive monitoring (e.g. for credit card numbers or message validity), since book_flight is a more important service.

3. The impacts of using a particular monitor will be similar, regardless of the service or quality type monitored. For example, the impact of using a probe type monitor
will usually be higher than the impact of using an **interceptor** type monitor, and this is independent of the quality type or service being monitored. In this example, a **probe** type monitor increases the load on the service by actively executes a service before analysing the results, whilst the **interceptor** type monitor only analyses existing service invocations, and so does not increase service load.

The set of **usage weights** is similar to the set of impact weights, except that instead of scaling the size of impacts, the usage weights scale the likelihood that an impact will occur. For example, there may be a higher likelihood that **response time** will be impacted than **security**. Usage weights model these relationships.

‘**Closeness**’ is used to model systems that have interdependencies between services, and services with varying levels of coupling, interaction, or impact on each other. For example, monitoring the service “verify credit card” should have a larger impact on the service “purchase book” than it does on the service “list books”, since “purchase book” requires the use of “verify credit card”, but “list books” does not. Furthermore, monitoring a service has more impact on that service than any other - i.e. monitoring “verify credit card” has a higher impact on “verify credit card” than it does on “purchase book” (remembering that impacts are measured as a percentage of the QoS of each service).

For these randomly generated cases, closeness of services is measured by the service names - services are named \( s_0 \ldots s_n \), and closeness is \([n - \text{absolute}(s_a \text{ numeral} - s_b \text{ numeral})]\), for services \( s_a \) and \( s_b \). This closeness for each \( s_a/s_b \) pair is used as a multiplier for **Impact Levels**. Additionally, for \( s_a \) and \( s_b \), if \( a = b \) (meaning the service to be impacted is the service being monitored), then the closeness value is further increased.

**Algorithm 5.5.2 Trended Random Impacts Generator**

1. for all (m,s,q,ml) in Monitoring_Capabilities_File do
2. append( Monitoring_Capabilities{m,s,q}, ml )
3. end for
4. for all q in Quality_Types[] do
5. Quality_Weights{q} ← rand(1)
6. Quality_Impacting_Likelihood{q} ← rand(1)
7. Quality_Impacted_Likelihood{q} ← rand(1)
for all s in Services[] do
    Service_Weights[s] ← rand(1)
    Service_Impacting_Likelihood[s] ← rand(1)
    Service_Impacted_Likelihood[s] ← rand(1)
end for

for all m in Monitors[] do
    Monitor_Instance_Weights[m] ← rand(1)
    Monitor_Overhead_Weights[m] ← rand(1)
    Monitor_Overhead_Likelihood[m] ← rand(1) × overhead_modifier
    Monitor_Instance_Likelihood[m] ← rand(1) × instance_modifier
end for

for all MC in Monitoring_Capabilities do
    if rand(1) ≤ Monitor_Instance_Likelihood{MC.m} × Quality_Impacting_Likelihood{MC.q} × Service_Impacting_Likelihood{MC.s} × instance_pr_modifier then
        for all s in Services do
            if rand(1) ≤ Service_Impacted_Likelihood{s} then
                for all q in QualityTYPES do
                    if rand(1) ≤ Quality_Impacted_Likelihood{q} then
                        impact_level ← 0
                        for all ml in Monitoring_Capabilities{MC.m,s,q} do
                            if ml == 0 then
                                impact_level ← 0
                            end if
                        end for
                        append(Impacts[], (MC.s, MC.q, MC.m, ml, q, s, impact_level))
                        impact_level ← impact_level + rand(n - abs(sa-sb)) / divisor
                        impact_level ← impact_level × Quality_Weights{MC.q} × Service_Weights{MC.s} × Monitor_Instance_Weights{MC.m}
                    end if
                end for
            end if
        end for
    end if
end for
for all \( m \) in Monitors do

if \( \text{rand}(1) \leq \text{Monitor.Overhead.Likelihood}\{m\} \) then

for all \( s \) in Services do

if \( \text{rand}(1) \leq \text{Service.Impacted.Likelihood}\{s\} \) then

for all \( q \) in Quality.Types do

if \( \text{rand}(1) \leq \text{Quality.Impacted.Likelihood}\{q\} \) then

impact.level \( \leftarrow \text{rand}((n - \text{abs}(sa-sb)) / \text{divisor}) \)

impact.level \( \leftarrow \text{impact.level} \times \text{Quality.Weights}\{q\} \times \text{Service.}

\text{Weights}\{s\} \times \text{Monitor.Overhead.Weights}\{m\} \)

append(\text{Overheads[]}\), \( (m,s,q,\text{impact.level}) \))

end if

end for

end if

end for

end if

end for

end if

end for

end if

end for

end if

end for

end for

print \text{Impacts[]}\]

print \text{Overheads[]}\]

Algorithm 5.5.2 shows the Trended Random Impacts Generator, which extends the Pure Random Impacts Generator. The difference between the two is the generation of Quality Type weights, Service weights, and Monitor weights.

The Quality Type weights are assigned to each quality type (The Quality.Weights in Line 5). These increase or decrease the impact level of each quality type (e.g. this reflects that monitoring security usually has a greater impact than monitoring response time). We also generate Quality.Impacting.Likelihood weights for each quality type (Line 6).

These Quality.Impacting.Likelihood increase or decrease the likelihood of monitoring
of a quality type leading to an impact (e.g. this reflects that monitoring security is more likely to have an impact than monitoring response time, regardless of the scale of that impact). Finally, we generate Quality_Impacted_Likelihood weights for each quality type (Line 7). These Quality_Impacted_Likelihood weights increase or decrease the likelihood that a quality type will be impacted by monitoring (e.g. this reflects that response time is more likely to be impacted than security when performing monitoring activities). Together, these three weights are used to meet the first assumption above.

Similarly to quality types, we generate impact weights for services (Lines 9-12), representing assumption 2, and monitors (Lines 14-18), representing assumption 3.

All of the generated impact weights are used to scale the likelihood (Line 21) and size (Line 33) of Monitoring Setting Impacts, as well as the likelihood (Lines 38, 40, and 42) and size (Line 44) of Overhead Impacts.

The closeness-based scaling occurs on Lines 32 and 43, where the limit for the rand function is the closeness of the two services concerned (i.e. that service being monitored and that service being impacted).

Modifiers for the probability of Overhead Impacts and monitoring impacts are taken as parameters, so that the average amount of impact can be varied.

5.5.3 Utility Function Generators

The Utility Function Generators take as input a Monitoring Capabilities File and a Utility Likelihood, and output a Utilities File containing Utility Functions based on the monitoring capabilities in the Monitoring Capabilities File. The Utility Likelihood describes the probability that a utility will be attached to some (quality type/service) pair.

The output Utilities File has columns of [service/quality type/monitoring level/Utility], and each row describes the Utility of measuring a particular quality type of a particular service at a certain monitoring level or higher.

5.5.3.1 Pure Random Utility Function Generator

The Pure Random Utility Function Generator extracts all monitoring capabilities from a Monitoring Capabilities File, and outputs a set of utilities based on these.
Utility values are randomly generated according to a supplied random cutoff. For a cutoff of 50%, on average 50% of service/quality type pairs will have some utility associated with them.

**Algorithm 3** Pure Random Utility Function Generator

1. for all s in Services[] do
2.   for all q in Quality_Types[] do
3.     if rand(1) ≤ utility_likelihood then
4.        max_utility ← rand(1)
5.        for all ql in Quality_Levels[] do
6.           for all ml in Monitoring_Levels[] do
7.              if ml == 0 then
8.                 util ← 0
9.              else if ml == 1 then
10.             util ← ql × max_utility
11.            else
12.             util ← ql × rand(max_utility)
13.         end if
14.         append(Utilities[], (s,q,ql,ml,util))
15.     end for
16.   end for
17. end if
18. end for
19. end for
20. print Utilities[]

Algorithm 3 shows the **Pure Random Utility Function Generator**. Each Quality_Type (Line 2) is enumerated for each Service (Line 1). If the utility_likelihood cutoff is met (Line 3), a value is generated for the maximum utility of this Service/Quality Type pair (Line 4). Then, for each Quality_Level (Line 5) and Monitoring_Level (Line 6), a Utility Function is created. If the Monitoring Level is 0, then so is the utility level (Lines 7-8), since we assume that not monitoring something provides no value. If the Monitoring Level is 1 (100%), then the utility level is a
product of the quality level $q_l$ and the max.utility for this Service/Quality Type pair (Lines 9-10). If the Monitoring Level is between 0 and 1, then the utility is a product of the quality level $q_l$ and some random value up to max.utility (Lines 11-12). Each utility value is then appended to the set of Utilities (Line 14), and finally the set of all Utilities is printed at Line 20.

For any (Service, Quality) pair that meets the utility_likelihood condition, a utility value is output. That utility value will be 0 if the Monitoring Level is 0 (Line 8), a function of max.utility and the current quality level $q_l$ if the Monitoring Level is 1 (Line 10), or a function of a random value up to max.utility and the current quality level $q_l$ if the Monitoring Level is less than 1 (Line 12). This means that the utility value will increase (or at least not decrease) as quality levels and Monitoring Levels increase, as would be expected.

5.5.3.2 Trended Random Utility Function Generator

The Trended Random Utilities Generator extends the Pure Random Utilities Generator by using random weights for both the probabilities of a Utility Function existing for each quality type and service, and the size of the utility for each quality type and service.

The Trended Random Utility Generator scales the size of the Utility Functions, and likelihood of a Utility Function existing for each quality type and service in the same way as the Trended Random Impacts Generator. This is based on the assumption that if a service has a high utility for one quality type (relative to other services and the same quality type), then it is likely to have a high utility for other quality types as well. For example, the book_flight service will usually have higher utility than flight_info, independent of the quality type.

Similarly, if a quality type has higher utility for one service (relative to other quality types), then it is likely to have higher utility for other services as well. For example, reliability will usually have a relatively higher utility than response time, independent of the service being monitored.

Algorithm 4 shows the changes to the Trended Random Utilities Generator from the Pure Random Utilities Generator. These changes are weights
Algorithm 4 Trended Random Utility Function Generator

1: for all q in Quality_Types[] do
2:   Quality_Weights{q} ← rand(1)
3:   Quality_Usage_Weights{q} ← rand(1)
4: end for
5: for all s in Services[] do
6:   Service_Weights{s} ← rand(1)
7:   Service_Usage_Weights{s} ← rand(1)
8: end for
9: for all s in Services[] do
10:   for all q in Quality_Types[] do
11:     if rand(1) ≤ Service_Usage_Weights{s} and rand(1) ≤ Quality_Usage_Weights{q} then
12:       max_utility ← rand(1) × Service_Weights{s} × Quality_Weights{q}
13:       for all ql in Quality_Llevels[] do
14:         for all ml in Monitoring_Llevels[] do
15:           if ml == 0 then
16:             util ← 0
17:           else if ml ≥ 1 then
18:             util ← ql × max_utility
19:           else
20:             util ← ql × rand(max_utility)
21:           end if
22:         append(Utilities[], (s,q,ql,ml,util))
23:       end for
24:     end if
25:   end for
26: end for
27: end for
28: print Utilities[]
for Quality.Types (Line 2) and Quality.Usage (Line 3), and Services (Line 6) and Service.Usage (Line 7). These weights allow for an output that is more likely to have quality types and services with similar utility values. Line 11 shows the use of the Service.-Usage.Weights and Quality.Usage.Weights, and Line 12 shows the new utility value scaling based on the Service.Weights and Quality.Weights.

5.6 Evaluation of the Optimisation Effectiveness

We now present a set of performance experiments and results designed to quantify the benefits of our monitoring optimisation technique. These experiments use an implementation of the iTravel case study as well as randomly generated problem sets.

For any Web Service monitoring system, there exists one or more optimal monitoring configurations (yielding the highest utility) in terms of monitoring coverage and Web Service performance. The purpose of these tests is to discover these optimal monitoring configurations, and compare the utility and performance of each optimal configuration to the utility and performance of each corresponding maximum (un-optimised) monitoring configuration, in which all monitors run at 100%.

5.6.1 Randomly Generated Experiments

To test the solution quality of our optimisation system for a wide variety of systems, we have designed ten classes of system, each with a different number of services, monitors, and quality types, different amounts of monitoring overlap, levels of monitoring impact, and levels of monitoring value. These classes allow us to cover a broad range of system types, within the limitations imposed by the complexity of the enumerative optimiser.

A set of search spaces were generated using the TRENDED RANDOM GENERATORS. A total of 10 parameter sets for the trended random generators were used, giving 10 classes of search space. For each class of search space, we generated 5 systems for optimisation. For each of these 50 systems, the optimal result (from the ENUMERATIVE SOLVER) and result from max monitoring were recorded. In addition, the total number of possible combinations for each input set (the search space height) is reported.

Table 5.3 shows the search space parameters for each class a to j (described below).
Table 5.3: Trended Random Search Space Parameters

<table>
<thead>
<tr>
<th>type</th>
<th>Number</th>
<th>TR Monitor Generator</th>
<th>Probability Repeated</th>
<th>TR Cost Generator</th>
<th>TR Utility Generator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s  m  q  ml</td>
<td>p(s)  p(q)  p(ml)  p(q-r)  p(s-r)  p(ml-r)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>5  5  5  3</td>
<td>0.5  0.5  0.3  0.9  0.6  0.95</td>
<td></td>
<td>.2  .5</td>
<td>3  3  0.5</td>
</tr>
<tr>
<td>b</td>
<td>3  3  3  3</td>
<td>0.5  0.5  0.3  0.9  0.6  0.95</td>
<td></td>
<td>.2  .5</td>
<td>3  3  0.5</td>
</tr>
<tr>
<td>c</td>
<td>8  8  8  3</td>
<td>0.25 0.25 0.25 0.9  0.6  0.95</td>
<td></td>
<td>.2  .5</td>
<td>3  3  0.5</td>
</tr>
<tr>
<td>d</td>
<td>5  5  5  3</td>
<td>0.4  0.4  0.4  0.3  0.3  0.3</td>
<td></td>
<td>.2  .5</td>
<td>3  3  0.5</td>
</tr>
<tr>
<td>e</td>
<td>5  5  5  3</td>
<td>0.4  0.4  0.3  0.9  0.6  0.95</td>
<td></td>
<td>.2  .5</td>
<td>10  10  0.8</td>
</tr>
<tr>
<td>f</td>
<td>5  5  5  3</td>
<td>0.4  0.4  0.3  0.9  0.6  0.95</td>
<td></td>
<td>0.9  .9</td>
<td>10  10  0.8</td>
</tr>
<tr>
<td>g</td>
<td>5  5  5  3</td>
<td>0.4  0.4  0.5  0.3  0.3  0.3</td>
<td></td>
<td>.9  .9</td>
<td>10  10  0.8</td>
</tr>
<tr>
<td>h</td>
<td>3  3  3  10</td>
<td>0.5  0.5  0.3  0.9  0.6  0.95</td>
<td></td>
<td>.2  .5</td>
<td>3  3  0.5</td>
</tr>
<tr>
<td>i</td>
<td>3  3  3  5</td>
<td>0.5  0.5  0.5  0.9  0.6  0.95</td>
<td></td>
<td>.2  .5</td>
<td>3  3  0.5</td>
</tr>
<tr>
<td>j</td>
<td>5  5  5  2</td>
<td>0.5  0.5  0.5  0.9  0.6  0.95</td>
<td></td>
<td>.2  .5</td>
<td>3  3  0.5</td>
</tr>
</tbody>
</table>
Table 5.3 is divided into sections for each Trended Random Generator tools (TR Monitor Generator is the TRENDED RANDOM MONITORING CAPABILITY GENERATOR, TR Cost Generator is the TRENDED RANDOM COSTS GENERATOR, and TR Utility Generator is the TRENDED RANDOM UTILITY FUNCTION GENERATOR).

The Trended Random Monitoring Generator parameters are the number of services $s$, number of monitors $m$, number of quality types $q$, number of monitoring levels $ml$, probability that a service will be selected for monitoring $p(s)$, probability that a quality type will be selected for monitoring, $p(q)$, probability that a monitoring level will be selected for monitoring $p(ml)$, probability that a quality type will be monitored a second or more time (repeated) by the same monitor $p(q-r)$, probability that a service will be monitored a second or more time by the same monitor $p(s-r)$, and probability that a monitoring level will be used by a monitor for a quality type and service if that monitor is already enabled for that quality type and service at a different monitoring level $p(ml-r)$.

The Trended Random Cost Generator parameters are the probability that a monitor will have some overhead costs associated with it $p(oh)$, and probability that a monitoring capability will have some monitoring setting impacts associated with it $p(msi)$.

The Trended Random Utility Generator parameters are the number of quality levels for each Utility Function $ql$, number of monitoring levels for each Utility Function $ml$, and probability that a Utility Function will be assigned to each quality type/service pair $p(util)$.

For each class $a$ to $j$ in Table 5.3, the important distinguishing characteristics are highlighted with a grey background. Class $a$ is the default from which other parameter sets are derived. Class $b$ has a smaller search space through less services, monitors, and quality types. Class $c$ has a larger number of services, monitors, and quality types; however, the probabilities of monitoring have been lowered so that the search space is small enough for the enumerative solver to search though in a few hours. Class $d$ decreases the probability that a service/quality type/monitoring level will be duplicated - this gives results closer to the pure random results. Class $e$ increases the number of Utility Functions, representing a greater number of requirements on the system. Class $f$ increases the number of Utility Functions and costs. Class $g$ increases the number of Utility Functions and costs,
and decreases the amount of repeated service/quality type/monitoring level monitoring capabilities. Class h, i, and j vary the amount of monitoring levels.

5.6.1.1 Results
Table 5.4: Trended Random Test Results

<table>
<thead>
<tr>
<th>Type</th>
<th>Search Space Size</th>
<th>Optimal Solution Utility</th>
<th>Default Solution Utility</th>
<th>%Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>9216</td>
<td>2.028</td>
<td>0.811</td>
<td>60</td>
</tr>
<tr>
<td>a</td>
<td>3072</td>
<td>1.513</td>
<td>1.513</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>41472</td>
<td>0.061</td>
<td>0.013</td>
<td>79</td>
</tr>
<tr>
<td>a</td>
<td>442368</td>
<td>0.617</td>
<td>0.178</td>
<td>71</td>
</tr>
<tr>
<td>a</td>
<td>1536</td>
<td>0.079</td>
<td>0.079</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>6144</td>
<td>1.097</td>
<td>0.528</td>
<td>52</td>
</tr>
<tr>
<td>b</td>
<td>64</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>96</td>
<td>0.753</td>
<td>0.743</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0.147</td>
<td>0.147</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>12</td>
<td>0.077</td>
<td>0.077</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>6</td>
<td>0.006</td>
<td>0.006</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>4608</td>
<td>1.348</td>
<td>1.266</td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>64</td>
<td>1.522</td>
<td>0.518</td>
<td>66</td>
</tr>
<tr>
<td>c</td>
<td>288</td>
<td>0.321</td>
<td>0.321</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>2048</td>
<td>0.653</td>
<td>0.083</td>
<td>87</td>
</tr>
<tr>
<td>c</td>
<td>9216</td>
<td>1.754</td>
<td>1.615</td>
<td>8</td>
</tr>
<tr>
<td>d</td>
<td>648</td>
<td>1.526</td>
<td>1.526</td>
<td>0</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Type</th>
<th>Search Space Size</th>
<th>Optimal Solution Utility</th>
<th>Default Solution Utility</th>
<th>%Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>1728</td>
<td>0.134</td>
<td>0.134</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>18432</td>
<td>0.977</td>
<td>0.818</td>
<td>16</td>
</tr>
<tr>
<td>d</td>
<td>6912</td>
<td>0.715</td>
<td>0.715</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>55296</td>
<td>2.342</td>
<td>1.415</td>
<td>40</td>
</tr>
<tr>
<td>e</td>
<td>3072</td>
<td>0.563</td>
<td>0.417</td>
<td>26</td>
</tr>
<tr>
<td>e</td>
<td>9216</td>
<td>7.952</td>
<td>1.549</td>
<td>81</td>
</tr>
<tr>
<td>e</td>
<td>256</td>
<td>1.96</td>
<td>1.001</td>
<td>49</td>
</tr>
<tr>
<td>e</td>
<td>1152</td>
<td>1.152</td>
<td>1.152</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>36</td>
<td>5.991</td>
<td>5.991</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>1536</td>
<td>0.58</td>
<td>0.13</td>
<td>78</td>
</tr>
<tr>
<td>f</td>
<td>6912</td>
<td>0.453</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>f</td>
<td>384</td>
<td>1.584</td>
<td>1.197</td>
<td>24</td>
</tr>
<tr>
<td>f</td>
<td>432</td>
<td>0.295</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>f</td>
<td>1728</td>
<td>1.014</td>
<td>1.014</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>98304</td>
<td>3.573</td>
<td>0.264</td>
<td>93</td>
</tr>
<tr>
<td>g</td>
<td>6144</td>
<td>2.588</td>
<td>0.931</td>
<td>64</td>
</tr>
<tr>
<td>g</td>
<td>93312</td>
<td>1.704</td>
<td>0.133</td>
<td>92</td>
</tr>
<tr>
<td>g</td>
<td>62208</td>
<td>13.567</td>
<td>12.907</td>
<td>5</td>
</tr>
<tr>
<td>g</td>
<td>20736</td>
<td>4.567</td>
<td>1.664</td>
<td>64</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Type</th>
<th>Search Space Size</th>
<th>Optimal Solution Utility</th>
<th>Default Solution Utility</th>
<th>%Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>46656</td>
<td>9.702</td>
<td>5.94</td>
<td>39</td>
</tr>
<tr>
<td>h</td>
<td>3024</td>
<td>0.741</td>
<td>0.652</td>
<td>12</td>
</tr>
<tr>
<td>h</td>
<td>45000</td>
<td>5.163</td>
<td>5.163</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>121500</td>
<td>0.009</td>
<td>0.009</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>72000</td>
<td>0.162</td>
<td>0.064</td>
<td>60</td>
</tr>
<tr>
<td>h</td>
<td>48384</td>
<td>0.143</td>
<td>0.101</td>
<td>29</td>
</tr>
<tr>
<td>i</td>
<td>13824</td>
<td>2.556</td>
<td>2.416</td>
<td>5</td>
</tr>
<tr>
<td>i</td>
<td>1296</td>
<td>0.14</td>
<td>0.018</td>
<td>87</td>
</tr>
<tr>
<td>i</td>
<td>3456</td>
<td>0.141</td>
<td>0.141</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>2304</td>
<td>1.527</td>
<td>1.527</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>34560</td>
<td>4.934</td>
<td>2.962</td>
<td>40</td>
</tr>
<tr>
<td>j</td>
<td>2048</td>
<td>2.102</td>
<td>2.04</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>16384</td>
<td>0.198</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>j</td>
<td>131072</td>
<td>1.262</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>j</td>
<td>262144</td>
<td>0.874</td>
<td>0.395</td>
<td>55</td>
</tr>
<tr>
<td>j</td>
<td>524288</td>
<td>1.751</td>
<td>0.599</td>
<td>66</td>
</tr>
</tbody>
</table>

Average: 36
Table 5.4 provides the results for these experiments. For each class \( a \) to \( j \), five systems were generated, and the results of optimising each system are each recorded as one row in Table 5.4. The table shows the net utility for each system under the optimal configuration and the maximum (default) monitoring configuration, as well the the difference between the utility values for the optimal and default configurations, as a percent.

Table 5.4 shows that optimisation increased net utility by 36\% over default monitoring, on average. Importantly, the additional utility received using optimal monitoring configurations increased with the number of parameters for that configuration (the search space size). This is demonstrated by Figure 5.4, which shows the difference in utility between the optimal and default monitoring configurations versus the size of the search space for the data in Table 5.4. Figure 5.4 shows that as the system complexity represented by search space size (horizontal axis) increases, the average percent increase in utility for optimised solutions (vertical axis) increases.

### 5.6.2 iTravel Experiment

The performance measurements are based on a series of simulated service executions. These simulations use response time measured through real service invocations to determine what the performance will be for a given monitoring configuration. Each experiment simulates 6,000 seconds of run-time. Using simulations in this manner allows us to measure the expected performance of hundreds of monitoring configurations over long run-times in a few seconds. Sample results of simulations have been re-run as real test executions, and the response times of simulations were accurate to within 5\%, on average.

The requirements for the iTravel system have been translated into a set of utility functions. These were used along with measured monitoring impacts to perform optimisation on the iTravel monitoring system. The optimal monitoring configuration for iTravel was enabling both **Interceptor** monitors at 100\% for all Qualities of Service, and disabling all other monitors. Under this configuration, the predicted base level **response time** of both services is 2.75 seconds, and the net utility is 4.245 which is the maximum achievable utility for iTravel’s requirements and monitoring capabilities. The predicted **response times** under the default (maximum) monitoring configuration are 4.3 seconds, and the resulting net utility is also 4.245, as discussed in Section 5.4.6. Since we are presenting a
Figure 5.4: Enumerative Vs Default Monitoring
simple scenario, it is possible to select this optimal configuration manually by examining the system requirements and impacts. However, we intend for these optimisations to be applied to larger and more complex scenarios for which optimisation would be too difficult to achieve without automation.

5.6.2.1 Results

We report both the resulting average response times and utilities. Response time results demonstrate that the performance of the optimised system has increased compared to a baseline, unoptimised system. Utility results demonstrate that we have met iTravel’s monitoring requirements. In each test, the total utility actually increased, as only the minimum valuable monitoring level was met, allowing for a performance increase, minimising the chance of a penalty to consumers for overly long response times.

Figure 5.5 shows response time versus load level for the iTravel scenario with unmonitored, maximum, and optimal monitoring configurations. The load levels on the horizontal axis represent the average number of active client requests, whilst the vertical axis gives the average response time over both services.

Optimisation reduced the average response time of the iTravel scenario by approximately 30% from maximum monitoring (from 14.24 to 10.31 seconds on average), and reduced the average response time impact of monitoring by over 80% (the optimal monitoring configuration was on average 7% slower than no monitoring, versus the maximum monitoring configuration’s 43%). Our design-time optimisation in Section 5.4.6 predicted that the response time impact under the optimal configuration would be 8%, and the response time impact under the max configuration would be 42%. In each case, our predicted values were out by just 1%, when averaged over all load levels in the system.

The horizontal, dashed line on Figure 5.5 shows the 10-second boundary, for which penalties apply in the iTravel system (from requirements SLA1 and SLA2). The optimised monitoring configuration stays under this 10-second boundary for approximately twice as long as the maximum monitoring configuration, i.e. the system with optimised monitoring dealt with twice as much load before a penalty for slow response times would have been paid.

The utility provided by the unmonitored solution was 0, since no requirements could be
verified. Conversely, the Maximum and Optimal monitoring configurations both included complete monitoring coverage. For each of these two configurations, the utility at or below 10 seconds response time was 4.245, and the utility above 10 seconds response time was 3.75.

To verify that the values selected for our motivating scenario were realistic, we repeated all tests with randomly generated utility functions and monitoring impacts ranging from mild (less than .01% per monitoring instance on average) to severe (up to 20% per monitoring instance), under various load levels. Figure 5.6 shows the results of these experiments. As per Figure 5.5, the average response time at each client load level is shown for each of the optimal, max and null monitoring configurations. The response times for the optimised solutions were on average 40% lower than maximum monitoring, and the optimised response time impacts were on average 70% lower than maximum monitoring. Utility values for optimised systems were 30% higher than utility values for systems with default (maximum) monitoring, on average.
Figure 5.6: iTravel Response Times Under Randomised Utilities and Impact Levels

5.7 Summary

We have presented a framework and techniques for optimising the configuration of Web Service monitors, in order to maximise utility for a Web Services provider. We have described a set of tools for generating randomised optimisation problems that, together with our iTravel scenario, were used to measure the effectiveness of our technique over a wide range of possible application scenarios.

A prototype instantiation of our proposed framework was used for a series of tests, which demonstrated that both utility and performance can be improved by optimally configuring a Web Services monitoring system. The results of our experiments demonstrate that both the utility and performance of a Web Services monitoring system can be increased by optimising the configuration of Web Service monitors. Although the results cannot be generalised to any Web Service system as they depend on system properties, benefits can be substantial, even for a simple system. These benefits depend on the amount of overlapping functionality of a monitoring system, the level of impacts of monitors, the value of monitoring, and the value of achieving performance goals. As such, the benefit

157
of optimisation will increase as a Web Services system becomes more complex, with more overlapping monitors to choose from, more system elements to monitor, and a greater number of monitoring and performance requirements to meet.
Chapter 6

Run-Time Monitoring Optimisation

This chapter describes our run-time monitoring optimisation framework and associated techniques. We extend the design-time optimisation described in Chapter 5 to work in a run-time system, with automatic management and feedback controls. We also describe a heuristic algorithm for solving the optimisation problem to ensure that the run-time optimisation system is fast enough to be used continuously at run-time. The implementation of our run-time system and heuristic are both described, along with experiments on each. In Chapter 5, we demonstrated that by balancing the monitoring coverage and the monitoring impact on the Quality of Service (QoS) delivered, net business value of a service provider can be significantly increased. However, it included an enumerative approach that due to its exponential time complexity does not scale to large solutions, and cannot be continuously applied to a system at run-time as the system changes. Potential run-time changes include:

- Long-term changes to the number of consumers;

- Short-term or temporary changes in user load, requirements or quality goals such as those that may occur during promotional periods; and

- Long term, high-impact changes from new or modified business requirements (e.g. the project team may impose some new quality goals), new or modified Service Level Agreements (SLAs) (e.g. a new consumer ‘gold’ class may be added with stricter
requirements), changes to the set of available monitors, added or removed services, and changes that affect the QoS being delivered, such as modified infrastructure.

When these changes occur, a design-time optimisation may become suboptimal. The greater the changes, the further away from optimal the design-time optimisation is likely to be.

In order to ensure the configuration of a suite of monitors stays optimal, optimisation must be re-applied at run-time whenever certain system changes such as those identified above occur. This requires a method to detect those system changes, calculate a new optimal configuration, and apply it to the set of monitors, all at run-time. This optimisation system must be flexible enough to manage different types of monitors, and fast enough to calculate and apply a new optimal monitoring configuration at run-time.

In this chapter, we introduce a novel approach and supporting framework for run-time optimisation of Web Service (WS) monitoring configurations, which uses actual measured QoS and system changes to continuously re-optimise a suite of WS monitors. We have implemented our proposed framework, including an incremental Genetic Algorithm (GA)-based optimiser. We demonstrate that this optimiser is fast enough to be applied to large systems at run-time, and achieves close-to-ideal monitoring configurations, delivering significant benefit to the service provider in terms of meeting monitoring requirements and minimising monitoring impacts.

Section 6.1 extends the iTravel case study from Chapter 2. Section 6.2 defines the run-time optimisation problem. Section 6.3 describes our run-time optimisation management system. Section 6.4 describes our proposed heuristic optimisation technique. Section 6.5 describes a set of techniques to reduce the size of the monitoring optimisation search space. Section 6.6 describes a set of experiments used to determine the set of ideal parameters for our heuristic-based optimisation technique. Section 6.7 describes a set of experiments to measure the solution quality and performance of our heuristic-based optimisation technique. Section 6.8 describes a set of experiments used to measure the effectiveness of our run-time optimisation framework.
6.1 iTravel Case Study

Recall iTravel, which offers Web Services to travel agencies for booking flights, as introduced in Chapter 2. iTravel has a service `flight_info`, used to lookup flight information, and a service `book_flight`, used to book flights. iTravel has SLAs with its consumers, giving guarantees for response time, privacy and reliability. If iTravel is unable to demonstrate that they have met these requirements, they must refund affected consumers a percentage of their monthly account fees. iTravel must also meet its corporate governance requirements which make it accountable for ensuring the security and privacy of its customers’ information. iTravel is liable for large fines, damages and reparation if they cannot demonstrate that these requirements have been met. As part of meeting both the consumer and corporate governance requirements, iTravel monitors its Web Services.

6.1.1 Recent System Changes

Although the original design-time optimisation yielded an optimal WS monitoring configuration for iTravel at design-time (as described in Chapter 5), aspects of the iTravel system have since changed.

6.1.1.1 Requirements changes

iTravel now offers their service to the public, which means that on top of their 100 travel agency consumers, they have many thousands of public consumers. Since these public consumers receive the service without an account fee, they do not receive an SLA with monetary penalties. However, to retain customers the iTravel project team have imposed
service level goals for the publicly available services as retaining such public customers through satisfactory QoS is regarded as delivering particular business value (e.g. through advertising revenue). These are that:

G1. flight_info reliability will be ≥ 95%, value=$0.5 Per Consumer

G2. book_flight reliability will be ≥ 98%, value=$1.0 Per Consumer

G3. book_flight response time will be ≤ 2s, value=$1.0 Per Consumer

G4. book_flight response time will be ≤ 2.5s, value=$0.75 Per Consumer

G5. book_flight response time will be ≤ 4s, value=$0.50 Per Consumer

G6. book_flight response time will be ≤ 8s, value=$0.25 Per Consumer

G7. flight_info response time will be ≤ 2s, value=$1.0 Per Consumer

G8. flight_info response time will be ≤ 2.5s, value=$0.75 Per Consumer

G9. flight_info response time will be ≤ 4s, value=$0.50 Per Consumer

G10. flight_info response time will be ≤ 8s, value=$0.25 Per Consumer

The response time goals G3 to G10 are graded, and represent the decreasing value to iTriavel as response times increase. Note that these requirements are not additive (e.g. the value of achieving response time under 2 seconds for book_flight is $1.0 per consumer, not $1.0+$0.75+$0.5+$0.25. Although the penalties for the goals G1 to G10 are not actually paid out by iTriavel, they represent the value to iTriavel of achieving those service levels. The value is calculated per consumer, and the number of consumers is estimated as the average number of monthly public users, which for iTriavel is currently 5,000.

6.1.1.2 Baseline performance changes

iTriavel has also modified its hardware configurations. Due to these changes, the baseline QoS has changed. Baseline response time of both services has decreased to 2 seconds, and reliability of both services is now 99%. Privacy and security remain unchanged.
6.1.1.3 Monitors and monitoring impact changes

The probe-type monitors have been modified and no longer impact the reliability of services. Additionally, analysis has revealed that the interceptor-type monitors have a 10% impact on response time of both services - i.e., the `flight_info` interceptor and `book_flight` interceptor each impact both the `flight_info` and `book_flight` services by 10%. This is due to changes in the network configuration, required to meet expected higher loads. Finally, each interceptor monitor now impacts response time of the service that it is monitoring by a further 2%, per quality type measured.

6.1.2 Design-Time Optimisation Validity

Due to the changes to requirements, hardware, and monitoring impacts described above, the delivered QoS and delivered utility of the original design-time optimisation have changed. Under the previous monitoring configuration, the net utility is 4.912, out of a potential 5.995.

The response times are just over 2.5 seconds, and reliability of both services is 98%. This is due to the longer response times of using the interceptor monitors. The re-application of the design-time optimisation to the new changed system situation gives a new, optimal configuration, involving using both probe monitors at 100% for all quality types, and disabling all other monitors. This gives a response time of approximately 2.2 seconds and a reliability of 99%, for both services. The net utility of this new optimal monitoring configuration increases by approximately 7%, to 5.288, which represents a business value increase of $14,000. Note that the theoretical potential of 5.995 cannot be reached with the current system characteristics - 5.288 is the highest achievable utility. This is because the requirements G3 and G7, stating that the response time of the `book_flight` and `flight_info` services will be less than two seconds cannot be met when performing any other monitoring (since the baseline response times for these services is already two seconds). Our optimal configuration increases the response times of the `book_flight` and `flight_info` services by 16% to approximately 2.4 seconds, and meets all other monitoring goals. As such, the optimal configuration meets every requirement except G3 and G7.
6.1.3 Future System Changes

The above scenario has demonstrated that it is beneficial to re-optimise a monitoring system when changes are made after the initial design. However, iTravel expects that changes will become more frequent, due to their services being publicly available. iTravel is unable to predict the long-term loads on their services. As such, QoS levels will vary as loads increase or decrease, and hardware is modified to suit. iTravel expects to add up to five new services over the next six months, as well as making changes to system monitors to account for these new services. Whilst these system-level changes may be accounted for using design-time optimisation, iTravel periodically runs promotions for specific services, and wishes to increase the QoS delivered for those services during the promotion periods (which may be between a few hours and a week). These changes require a method to perform optimisation faster than the design-time optimiser can perform.

Due to these recent and future changes, iTravel requires a method for automatically re-optimising its monitoring configuration at run-time as often as the changes dictate. This method should be fast enough to find a good (better than default\(^1\)) monitoring configuration within one hour, and should be capable of automatically applying that configuration to the iTravel monitoring system. The method is required to automatically re-optimise the system if the delivered QoS is not meeting original expectations, or some other system parameter such as the number of consumers changes.

6.2 Run-Time Optimisation Problem Description

The optimisation problem is to maintain a run-time system with a monitoring configuration that gives the maximum value in terms of QoS impact (costs) and monitoring coverage (benefits). To achieve this, we have three important requirements. Firstly, the recently recorded QoS levels for the system are required, in order to ensure the optimisation remains accurate. Secondly, the entire process from information gathering, to computing an optimal solution and applying that solution should be automated so as to not require constant human intervention. Thirdly, the method for computing an optimal configuration

\(^{1}\)‘Default’ monitoring is monitoring without any optimisation, i.e. all monitors are left on at 100% monitoring levels. This is referred to as ‘maximum’ or ‘max’ monitoring.
should be fast enough to find a solution for a large system in a reasonable time (which we consider is under one hour).

In the following sections, we present our optimisation framework involving a set of components that are used at run-time to gather information (including changes) on the system, calculate new optimal monitoring configurations, and apply them to the monitoring system. We also describe our Genetic Algorithm-based optimisation routine, which allows for optimisation of large systems within desired time-frames.

6.3 Run-Time Management System

The run-time optimisation framework is shown in Figure 6.2, and involves an Optimisation Routine, a Run-Time Controller, a Configuration Manager, a set of Monitoring Managers, and a QoS Analyser. In effect, these components form a feedback control loop: the Monitors (together with the services they monitor) are the target system to be controlled, the Run-time Controller and the Optimisation Routine form the controller, the Configuration Manager and Monitoring Managers form the actuator, and the Monitor Managers and QoS Analyser form the sensor. The set point for this control system is derived from the set of utility functions for monitoring coverage and QoS, the control input (i.e. output from the controller) is the optimal monitoring configuration, and the measured system output is the actual measured QoS.

The initial baseline system performance and descriptions of the monitors and their QoS impacts are initial characterisations of the target system. This information is used for the design of and decision making in the controller (i.e. the Run-time Controller and Optimisation Routine).

As such, the run-time management framework is set to cope with changes to the system and its requirements, and monitor set-ups and even changes to the actual system QoS caused by unknown or un-modelled factors (e.g. load from external services), and calculate a new optimal monitoring configuration. Below, we describe each of the components of this framework.
6.3.1 Run-Time Controller

The Run-time Controller is responsible for gathering all Optimisation Parameters, sending these to the Optimisation Routine, and sending the resulting Optimal Monitoring Configuration to the Configuration Manager for application.

The Optimisation Parameters consist of:

- The sets of QoS Requirements on the services, which are sourced from SLAs, corporate governance, or other sources, as described in Chapter 2. These requirements are translated into utility functions that describe the value received for monitoring a particular quality of a particular service, at a specified minimum monitoring level or higher, whilst achieving a specified minimum quality level or higher;

- The set of available Monitoring Capabilities, describing the monitors available,
and what qualities of what services they can monitor, and at what monitoring levels;

- The set of known Monitoring Impacts, describing the QoS impacts that are incurred for each monitoring level of each Monitoring Capability;

- The baseline system performance (i.e. the set of Ideal Quality Levels for all services and quality types without monitors enabled); and

- The recent actual QoS for each service and quality type as gathered by the Monitors and provided by the Monitoring Managers.

These parameters are used by the Optimisation Routine to generate an Optimal Monitoring Configuration, which is returned to the Run-time Controller. The Run-time Controller then forwards this Optimal Monitoring Configuration to the Configuration Manager.

After an initial optimisation, the Run-time Controller waits for changes to any of the parameters listed above. Currently, changes in system states such as available services or SLAs are detected and made available by monitoring system description files. However, this may be automated if system components broadcast their descriptions during deployment or modification. When a significant change is detected, the Run-time Controller sends the new set of Optimisation Parameters to the Optimisation Routine to obtain a new Optimal Monitoring Configuration. Whilst the definition of a ‘significant change’ may vary depending on user perspective, we consider a change in average QoS levels of over 5%, or any system change such as a new monitoring capability, a change to monitoring impacts, or a new service as ‘significant’.

6.3.2 Optimisation Routine

The Optimisation Routine takes a set of Optimisation Parameters, and uses these to determine an Optimal Monitoring Configuration. An Optimal Monitoring Configuration is a complete set of monitoring settings (i.e., all monitoring capabilities in the system are assigned some monitoring level) that yields the greatest utility for the Web Service provider, according to the set of provided Utility Functions. In general,
an Optimisation Routine will search through the space of possible monitoring configurations, scoring each based on the predicted QoS levels that will be achieved under it (after accounting for QoS impacts from monitoring) and the monitoring coverage that it provides. Details of our enumerative approach to optimisation are provided in Chapter 5. Below, we present details of a faster, heuristic optimisation routine.

6.3.3 Configuration Manager

The Configuration Manager maintains a database of available Monitors as well as a translation lookup table, in order to send individual Monitoring Settings to each Monitor. The translation lookup table allows for basic translation of terms that may be monitor specific, e.g. from “delay” to “response time”.

The Configuration Manager takes requests for either monitoring results information, or monitoring reconfiguration instructions. Although these requests may come from any source, in our case they come from a central Run-time Controller.

When the Configuration Manager receives a request for monitoring results information (not shown in Figure 6.2), it sends the request to the appropriate Monitoring Managers, which request Measurements from the Monitors, and return those Measurements to the QoS Analyser for aggregation.

When the Configuration Manager receives a new Optimal Monitoring Configuration, it looks up a database to determine the location of the Monitoring Manager for each Monitor. The Configuration Manager then sends Monitoring Settings in a generic form to each Monitoring Manager, as appropriate. This generic request format is of the form "Monitor, Quality Type, Service Monitored, Monitoring Level", specifying which monitor should be set to observe which quality type of which service at what monitoring level.

6.3.4 Monitoring Managers

Management of monitors involves configuring monitors at design-time or run-time, in order to achieve a desired monitoring configuration. Not only are monitors geographically separated, they may be of different types. Since a monitor may be anything from a hardware router to a piece of custom software, a management system must be able to deal
with monitors with different interfaces. We achieve this with Monitoring Managers, which translate between monitor-specific languages and a generic syntax used by the central Configuration Manager.

Monitoring Managers act as an interface between the generic optimisation components in our system (the Optimisation Routine, the Runtime Controller, the Configuration Manager and the QoS Analyser) and the system-specific components (the Monitors).

Each Monitor in the system has a Monitoring Manager to act as an interface to the Configuration Manager and the QoS Analyser (there is a one-to-one relationship between Monitoring Mangers and Monitors). Monitoring Managers are responsible for applying Monitoring Settings on Monitors, and retrieving Measurements from Monitors, which are forwarded to the QoS Analyser.

Having a Monitoring Manager as an interface to each Monitor allows for Monitors of any type in the system, without the need to modify those Monitors to accept a standard set of configuration Commands or information requests for Measurements. Such Monitors may include hardware-based monitors such as network devices, and specific software monitors such as test probes and software firewalls.

The only requirement on Monitors is that they have some interface available that allows for their configuration at run-time, and allows for their monitoring results to be reported back to a Monitoring Manager. This flexibility allows for the use of existing, different classes of monitors in a target system, rather than requiring the use of only one type of monitor.

Below, we show the syntax for a monitoring configuration and logs for an example network monitor that measures response time and has a configuration file that has one setting per line:

```
percentPacketLog=50
enabled=true
targetIP=10.0.0.2
```

This monitor then outputs the results of monitoring in a log file in the following format:

```
1345883401::1345883405::10.0.0.2::217.98.15.5
1345883402::1345883404::10.0.0.2::104.24.9.14
```
where each line contains the Unix-formatted timestamp for the service request, followed by the timestamp for the service response, the IP address of the service and finally the IP address of the service consumer.

Our implementation uses a simple syntax for both sending Monitoring Settings to a Monitoring Manager (a comma-separated string containing a monitor ID, a service ID, a quality type ID and a monitoring level) and requesting monitoring Measurements from a Monitoring Manager (a comma-separated string containing a service ID, a quality type ID and a start and end time). The Monitoring Manager would translate between this generic syntax and the syntax accepted by the Monitor, as appropriate. For example, the Monitoring Manager would convert the monitoring setting request from NetworkRTMonitor,flightInfo,ResponseTime,0.5 to yield the example above with percentPacketLog=50, enabled=true, and targetIP=10.0.0.2. The Monitoring Manager would accept a measurement request flightInfo,ResponseTime,1345883401-1345883405 for Response Time of the flightInfo service across times 1345883401-1345883405 (these times align with the two lines in the log file in the example above) and return the result of flightInfo,ResponseTime,3 (the average between the two results shown in the log file for that monitor).

6.3.5 QoS Analyser

The QoS Analyser is responsible for gathering raw Measurements from Monitors via their Monitoring Managers, and aggregating those Measurements (over monitors and time) and returning a set of recent QoS levels achieved. Aggregating QoS levels over monitors allows for a particular QoS to be calculated as an average of a set of results from different monitors. Aggregating QoS levels over time allows for short, medium or long-term averages as well as the calculation of quality levels for long-lived properties (i.e., those that last for more than one service transaction) where necessary. The time period for which to compute measurements is set by the Run-time Controller.

The QoS Analyser listens for a Get_QoS SOAP request, containing parameters for:

- monitors, an array of monitors to retrieve QoS data from;
• **quality type**, the quality type of interest;

• **service**, the service of interest;

• **start time**, the most distant time from which measurements should be taken; and

• **end time**, the most recent time from which measurements should be taken.

The QoS Analyser then sends requests for monitoring results to each Monitoring Manager for each monitor in the monitors array, and averages the results that meet the **start time** and **end time** criteria. This averaged QoS result is then returned to the Run-time Controller.

### 6.3.6 Summary

To start, the Run-time Controller takes as input any system **Parameters** required for optimisation (requirements, monitoring impacts, baseline QoS, and monitoring capabilities), and sends these to an Optimisation Routine, which uses them to compute an Optimal Monitoring Configuration. The Run-time Controller then sends this Optimal Monitoring Configuration to the Configuration Manager. The Configuration Manager sends the Monitoring Settings for each Monitor to the corresponding Monitoring Manager. Each Monitoring Manager then sends monitor-specific **Commands** to the Monitor it is controlling to achieve the appropriate settings.

After the initial configuration is applied, the Run-time Controller waits for system changes that will trigger re-optimisation. These changes may be changes to the initial input **Parameters**, or changes in measured QoS. When either input parameters change or measured QoS changes by a threshold amount (e.g. 10%), the Run-time Controller re-invokes the Optimisation Routine with the new **Parameters**, and sends through the new Optimal Monitoring Configuration to the Configuration Manager.

After the initial configuration, the Run-time Controller also periodically requests system QoS measurements, via the Configuration Manager. These requests are sent by the Configuration Manager to each applicable Monitoring Manager, which translates the requests and gathers QoS measurement information. The Monitoring Managers then aggregate the QoS data for their monitor, and forward the Measurements
to the Configuration Manager. The Configuration Manager performs any necessary aggregation of the Monitoring Manager's results, and returns the measured QoS to the Run-time Controller.

The full implementation details of each component described above are provided in Section 8.2.

6.4 Heuristic Optimisation Method

Since the Enumerative Solver has an exponential time complexity a heuristic is required to solve large problem sets, especially for run-time optimisation for which optimisation may be applied often or even continuously. This section contains a description of the design and implementation of a Genetic Algorithm, which is a heuristic designed to replace or augment the enumerative algorithm. This GA-based optimisation routine (The GA Solver) takes the same inputs as the Enumerative Solver (i.e. the same search space), models these as a GA, and attempts to find an optimal configuration for a suite of monitors. We have implemented this heuristic, and compared it to the Enumerative Solver for performance (how quickly the heuristic finds a solution) and solution quality (how close to optimal that solution is), on a wide range of problem sets.

6.4.1 Heuristic Search Techniques

There are various classes of metaheuristic for search or optimisation. Most heuristics are stochastic, since they use statistical inference to model random variance in models or input sets. An important attribute of stochastic metaheuristics is that they reduce the likelihood of search being stuck in local minima or maxima. Families of stochastic heuristics include those based on simulated annealing, random sampling, hill climbing, and nature-inspired heuristics including swarm intelligence methods (e.g., particle swarm optimisation, ant colony optimisation), and evolutionary algorithms (e.g. genetic algorithms).

The size and shape of the search space, cost of search, frequency of search, value of search results, and computational resources available are important factors to consider when selecting a metaheuristic. If a search space is purely random, then random sampling may be most appropriate heuristic. Conversely, if a search space is strictly monotonic,
then binary search may be the most appropriate algorithm. If local minima or maxima are not significant, then a basic hill-climbing algorithm may suffice. Where local minima or maxima are significant, methods such as simulated annealing, or nature-inspired algorithms may help by preventing searching getting stuck in those local minima or maxima.

The search space for optimising the configuration of Web Service monitors is large, and contains significant minima and maxima, as demonstrated by Figure 6.3, which is the search space for a simple system with just two monitors. As such, the classes of meta-heuristics that are most appropriate are those including simulated annealing, evolutionary algorithms, particle swarm optimisation, and ant colony optimisation. The most distinguishing characteristic between swarm intelligence methods and evolutionary algorithms is the use of techniques such as crossover and mutation in evolutionary algorithms.

The shape of the search space for Web Service monitoring optimisation is well suited to evolutionary algorithms, since the search space is likely to have significant minima and maxima, and the parameters in the search space (monitoring instance settings) can be modelled to ensure choice between only functionally equivalent monitors[48]. Since a genetic algorithm will divide the search space along these dimensions, it allows for fast convergence to a good solution by selecting the individual genes (monitoring instance settings) that contribute to solutions with high fitness, whilst simultaneously selecting a chromosome (whole set of genes) that yields the net highest fitness. Furthermore, due to the complexity of calculating the utility of each point in the search space, the extra
overhead of a GA over computationally simpler algorithms such as hill-climbing techniques is comparatively low.

For the above reasons, it was decided that genetic algorithms would be used as the first heuristic algorithm to be trialled for optimisation. Since our goals for optimisation quality and timeliness were met by the GA Solver (as described in the following sections), other classes of heuristics have not been investigated. An introduction to Genetic Algorithms is provided in Appendix D.

6.4.2 Genetic Algorithm-based Solver

A GA-based optimiser (the GA Solver) has been developed that takes as input the available Monitors in the system and their Capabilities, Services in the system and their Ideal Quality Levels, Requirements on the system translated into Utility Functions, recently delivered QoS, and Monitoring Impacts. These inputs are modelled as a GA by the GA Solver, which attempts to find an Optimal Monitoring Configuration through the standard process of Initialisation, Evaluation, and repeated Crossover, Mutation and Evaluation. The Enumerative Solver always produces an optimal monitoring configuration for the given inputs, whilst the GA Solver performs a heuristic search for an optimal monitoring configuration, but is not guaranteed to always discover one. The GA Solver takes as input a search space and GA parameters, runs for a fixed number of generations or until a plateau is detected, and outputs the best candidate monitoring configuration found.

The GA Solver’s population models the Monitoring Levels that each Monitoring Capability can be set to. This means each gene can be set to one of \( 0 \ldots n \) Monitoring Levels representing the set of Monitoring Levels available for that Monitoring Capability. For a set of \( N \) Monitoring Capabilities, we have a chromosome of width \( N \), and each gene of that chromosome can be set to any of its Monitoring Capabilities’ Monitoring Levels. For example, consider a system with two monitors, \( m_1 \), which can measure quality \( q_1 \) of service \( s_1 \) at monitoring levels \( \{0,1\} \); and monitor \( m_2 \), which can measure \( q_1 \) of service \( s_1 \) at monitoring levels \( \{0,0.5,1\} \). In this case, the GA would have

\[ \text{For simplicity, we refer to the output of the GA Solver as an 'optimal solution', even though it may be sub-optimal.} \]
a chromosome with two genes (one per monitoring capability), with the first gene having potential values 0 and 1, and the second gene having potential values 0, 0.5 and 1, and there are six possible population outcomes: \((\{0,0\}, \{0,0.5\}, \{0,1\}, \{1,0\}, \{1,0.5\}, \{1,1\})\).

The fitness function for the GA is taken directly from the Enumerative Solver. It takes an individual (a Monitoring Configuration) as input, calculates the impact of this configuration, then calculates and returns the total Utility of the configuration under the resulting impact. The GA Solver differs from the Enumerative Solver in that the GA Solver does not prune any candidate solutions - this means that the GA Solver will consider solutions with overlapping monitoring (when multiple monitors observe the same quality of service, increasing their total impact but not increasing monitoring coverage). This is desired, since a solution with overlapping monitoring, although guaranteed to be less than optimal, may still be a good candidate solution, and may indeed be only one gene switch or crossover away from ideal.

6.4.2.1 Genetic Algorithm Parameters

Other than the search space definition, the GA Solver takes as input the maximum number of Generations to run for, a Mutation Rate, a Crossover Rate, a Crossover Function type, a variable defining a maximum Plateau Length, and a Population Size.

The maximum number of Generations provides a termination criterion by limiting the run-time of the algorithm. The maximum Plateau Length is also a termination criterion. If the fittest individual does not change for the specified number of generations (the Plateau Length), then the GA is terminated earlier than the specified maximum number of Generations.

The Mutation Rate describes the percent of elements of any given population member that will be randomly modified. For a Mutation Rate of 0.01, 1% of the parameters of each individual will be changed to some other value.

The Crossover Rate describes the percentage of population that will be crossed-over. For a Crossover Rate of 0.01, 1% of the chromosomes of each population will be selected for crossover by the selected crossover function.

The Crossover Functions that have been tested are Roulette Single Point, Roulette Two Point, Roulette Uniform, Tournament Single Point, Tournament Two Point, Tour-
nament Uniform, Random Single Point, Random Two Point and Random Uniform. Each of these is described in Appendix D.

The remaining variable for the GA is the Population Size, which represents the number of individuals per population.

To determine the best general configuration for our GA, we have evaluated the Mutation Rate, Crossover Rate, Crossover Function type, Population Size, and maximum Plateau Length, as described in Section 6.6.

6.5 Search Space Reduction

The size of the search space for our monitoring optimisation problem is based on the number of possible monitoring settings in the system, and (as discussed in Chapter 5) is bound by:

$$O(\text{avg}(|ML_{mcap}|)^{|MCAP|}),$$

where $|ML_{mcap}|$ is the number of monitoring levels of monitoring capabilities, and $|MCAP|$ is the number of monitoring capabilities.

To reduce the size of this search space and allow for larger systems to be optimised at run-time, we have developed two search space reduction techniques. These are a Search Space Pruner and a Search Space Divider. Each of these techniques is described below.

6.5.1 Search Space Pruner

The Search Space Pruner attempts to reduce the size of the monitoring optimisation search space by pruning out monitors, monitoring capabilities, and monitoring capabilities’ monitoring levels that are guaranteed to never be part of an optimal solution.

The Search Space Pruner prunes the monitoring search space in three ways. First, we prune monitoring capabilities that are always more expensive in terms of impact than the benefits achieved by using them. Second, we prune individual monitoring levels that don’t yield higher utility than those lower monitoring levels below them. Third, we prune monitors that are always more expensive than other monitors that can perform the same monitoring tasks. Each of these pruning techniques is described below.
Monitoring capability pruning searches for a monitoring capability such as “book-flight Probe monitoring the book_flight service for response time at monitoring levels 0.0, 0.5 or 1.0”, where performing that monitoring function at any of those monitoring levels has a cost greater than the impacts that will occur. For the above example, if the net value of monitoring response time of the book_flight service is $100, but the per-instance costs of monitoring it at some monitoring level greater than 0 using the book-flight Probe are at least $200, then that monitoring level of that monitoring capability can be pruned. If all monitoring levels greater than 0 are pruned, then the monitoring capability can be pruned as well. Finally, if all monitoring capabilities for a monitor have been pruned, then that monitor can be pruned as well.

Individual monitoring level pruning prunes monitoring levels of monitoring capabilities that don’t yield higher utility than those below them. For example, if the utility of using the book_flight Probe to monitor the book_flight service for response time at monitoring level 1.0 is not greater than the utility of using the book_flight Probe to monitor the book_flight service for response time at monitoring level 0.5, then monitoring level 1.0 may be pruned from this monitoring capability. This is based on the assumption that monitoring at a higher level (and therefore with a higher cost) is not desired, unless there is some utility in doing so.

Monitor pruning prunes entire monitors that are guaranteed to never be part of an optimal solution. If there exist two monitors m₁ and m₂, and the monitoring capabilities of m₁ are a subset of those in m₂, and the cost of using any monitoring setting in m₁ is higher than the cost of using the same monitoring setting in m₂, then the monitor m₁ may be pruned. For example, consider a monitor m₁ that can observe quality type q₁ of service s₁ at monitoring levels \{0, 1\} (and has no other monitoring capabilities) with an overhead impact of 0.5 on q₁ of s₁, and a per-instance impact of 0.1 when using monitoring level 1 on q₁ of s₁; and monitor m₂ that can observe q₁ of s₁ at monitoring levels \{0, 1\} (and may have other monitoring capabilities), has no overhead impacts, and a per-instance impact of 0.05 when using monitoring level 1 on q₁ of s₁. In this case, monitor m₁ is guaranteed to always be suboptimal, since m₂ has lower impacts for the same monitoring operation. For pruning to occur, all of monitor m₂’s costs must be lower than those of monitor m₁.
6.5.1.1 Search Space Pruning Evaluation

We have performed a series of experiments to determine the effectiveness of pruning monitoring configuration search spaces. We have generated a set of search spaces with between five and seven monitors, services, and quality types using our trended random generator tools. For each system size between five and seven, we generated nine search space types with varying levels of randomness, by varying the probability-based parameters from Table 5.3 from 10% to 90%. For each problem size/randomness pair, we generated five search spaces, for a total of 135 test cases.

We measured the size of each of the 135 test cases in terms of width, which represents the number of search space parameters (i.e. the number of monitoring capabilities’ monitoring levels) and height, which represents the number of possible monitoring configurations. After this, we pruned each of the test cases’ search spaces, and measured the width and height again. Figure 6.4 shows the pruned and unpruned sizes averaged over all test cases, for systems with system sizes of 5, 6, and 7 (representing the numbers of monitors, services, and quality types). For test cases of system size five (five monitors, services and quality types), pruning reduced the average search space size from $1.6 \times 10^{75}$ to $6.7 \times 10^{49}$, for system size six, pruning reduced the average search space size from $1 \times 10^{130}$ to $1.4 \times 10^{89}$, and for system size seven, the average reduction was from $2.9 \times 10^{206}$ to $1.6 \times 10^{133}$. Note that it is not meaningful to discuss these results in terms of percentage of a search space pruned due to the scale of the results (e.g. the percent of elements pruned for system size five is approximately 99.999999999999999999999999999999%).

Figure 6.5 shows the pruned and unpruned search space sizes for the same test cases as above, but divides the search spaces according to the probability that a monitor will observe the same service or quality type with multiple monitoring levels (Search Space Repetitiveness). For Figure 6.5, the values on the horizontal axis represent the probability of all of each following being true:

- A monitor monitoring a second quality type for a service it can monitor (for a different quality type);

- A monitor monitoring using a second monitoring level for a quality of service it can monitor; and
A monitor monitoring a second service for a quality type it can monitor (for a different service).

A value of 1 represents a 10% probability (e.g. if a monitor can monitor $q_1$ of $s_1$ with monitoring level 0.25, it has a 10% probability of being able to monitor $q_1$ of $s_1$ with a monitoring level of 0.50), whilst 9 represents a 90% probability. Full details of the parameters are discussed in Section 6.7.2.2. Figure 6.5 groups each set of tests according to their input size, and gives the pruned and unpruned search space size for each input size/repetitiveness pair. This allows for a comparison of the effectiveness of pruning with consideration for both problem size and structure (represented by repetitiveness). Figures 6.6 and 6.7 show that there is no significant correlation between search space repetitiveness and the amount of elements that can be pruned from a search space. Figure 6.6 shows the percent of parameters pruned for each level of monitoring repetition, for system sizes 5, 6, and 7. Figure 6.7 shows the same information, with the percent of parameters pruned averaged over the three system sizes. Simple linear regression in Figure 6.7 shows there is little to no relationship detected between monitoring repetition and percent of parameters pruned. This indicates that pruning should be equally effec-
<table>
<thead>
<tr>
<th>Search Space Repetitiveness</th>
<th>Unpruned</th>
<th>Pruned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.5: Pruned versus Unpruned System Sizes

tive on systems with a lot of small, single-use monitors (represented by low Search Space Repetitiveness values), and systems with fewer, multi-use monitors (represented by high Search Space Repetitiveness values). The lack of a correlation between repetitiveness and pruning effectiveness is due to pruning removing more monitors with fewer capabilities in those systems with low repetitiveness, and fewer monitors with more capabilities in systems with high repetitiveness, resulting in similar reductions in the size of search spaces.

### 6.5.2 Search Space Divider

The **Search Space Divider** attempts to divide the monitoring optimisation search space into independent problem spaces that can be optimised individually and recombined. This reduces the time required to search due to the exponential complexity of the search space. A search space can be divided if there are two independent subsets of that search space.
Figure 6.6: Impact of Monitoring Repetitiveness on Percent of Parameters Pruned

Figure 6.7: Average Impact of Monitoring Repetitiveness on Percent of Parameters Pruned
Table 6.1: Divisible Search Space

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Monitoring Overheads</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m_1 )</td>
<td>( {(s_1, q_1, 0.3)} )</td>
</tr>
<tr>
<td>( m_2 )</td>
<td>( {\varnothing} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Monitoring Setting</th>
<th>Monitoring Level</th>
<th>Monitoring Setting Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m_1, s_1, q_1 )</td>
<td>0.0</td>
<td>( {\varnothing} )</td>
</tr>
<tr>
<td>( m_1, s_1, q_1 )</td>
<td>1.0</td>
<td>( {(s_1, q_1, 0.1)} )</td>
</tr>
<tr>
<td>( m_2, s_2, q_1 )</td>
<td>0.0</td>
<td>( {\varnothing} )</td>
</tr>
<tr>
<td>( m_2, s_2, q_1 )</td>
<td>1.0</td>
<td>( {(s_2, q_1, 0.2), (s_1, q_2, 0.2)} )</td>
</tr>
</tbody>
</table>

which:

(1) Have no overlapping monitoring, and

(2) Have no overlapping monitoring impacts.

Rule (1) is required, because if two or more monitors can observe the same target (i.e., they have the same monitoring capabilities), then a choice must be made between those monitors. Rule (2) is required, because if two or more monitors can impact the same target, then the impact of those monitors must be considered simultaneously. This is because, if two subsets are optimised independently (even if their monitors obey Rule 1), then the impacts of those subsets may combine to breach some SLA, and reduce the utility to a suboptimal level.

Consider the monitoring system with the monitoring capabilities and impacts described in Table 6.1. For this search space, monitors \( m_1 \) and \( m_2 \) are independent, because they do not monitor or impact the same QoS. Note that, even though monitor \( m_2 \) impacts service \( s_1 \), this is a different quality type \( (q_2) \) than that impacted by monitor \( m_1 \) \( (q_1) \), and so can be considered independently. The Search Space Divider takes as input the sets of monitoring capabilities and monitoring impacts, and outputs a set of independent subsets of those. These independent subsets are then optimised independently and re-combined to create a complete, optimal monitoring configuration.
6.5.2.1 Search Space Division Evaluation

We have performed a series of experiments to determine the potential effectiveness of search space division. We have generated a set of search spaces with between 10 and 30 monitors, services and quality types, and search space repetitiveness values between 1 and 9. These search space parameters were selected to generate large systems, since these are the target for search space division. For each system size between 10 and 30, and each search space repetitiveness value between 1 and 9, we generated 5 test cases, and executed the Search Space Divider on each, recording the total number of search spaces that each original search space could be divided into.

Figures 6.8, 6.9 and 6.10 show the results of our experiments on search space division. Figure 6.8 shows the system size value used to generate test cases versus the average number of subsets that system size could be divided into. It can be seen that as the system size increases, so does the average number of subsets. This is to be expected, since the larger a search space is, the more likely there is to be a unique subset within that search space. Figure 6.9 shows the search space repetitiveness value used to generate test cases versus the average number of subsets the resulting search spaces could be divided into. It can be seen that once search space repetitiveness reaches level 3, the average number of subsets increases before levelling off at approximately 2.5. Figure 6.10 shows a combination of both system size and search space repetitiveness versus numbers of subsets. This figure confirms that as system size and search space repetitiveness increase, so does the number of subsets that a search space can be divided into.

The average search space size used for these experiments was $1 \times 10^{302}$. This was reduced to an average of $1 \times 10^{280}$ when search space division was performed. Whilst the reduced search space size is still large, performing search space division is fast and so is still worthwhile to perform. Furthermore, our results indicate that, as expected, search spaces become more divisible as their structure and size increase.

6.6 Selection of GA Parameters

We have performed a series of experiments in order to determine ideal GA parameters. The following sections provide the results of evaluating the Mutation Rate, Crossover
Figure 6.8: Search Space Divisibility by System Size

Figure 6.9: Search Space Divisibility by System Repetitiveness
Figure 6.10: Search Space Divisibility by System Size and System Repetitiveness

Rate, Crossover Type, Population Size, Plateau Length, and Genetic Memory for the GA Solver. For each set of experiments, a range of values were selected and tested on 10 randomly generated systems. For all experiments, the default parameter settings were a Mutation Rate of 0.1, a Crossover Rate of 0.90, a Crossover Type of Tournament Single Point, a Population Size of 90 and no Genetic Memory (since all other parameters do not require multiple executions). These are common, median baseline settings for a GA[53]. For each experiment, these default settings were used whilst one parameter was varied. Unless otherwise stated, each experiment was run for 500 generations, and the resulting utilities were recorded.

6.6.1 Mutation Rate

We have measured the effectiveness of Mutation Rates of between 0.5% and 40%, with the results depicted in Figure 6.11. Note that we performed higher resolution tests below 10%, as Mutation Rates in the range of 1%-5% are generally considered good default values for GAs[53]. As demonstrated in Figure 6.11, the ideal Mutation Rate is 10% for our problem sets. However, the ideal Mutation Rate will vary depending on the system being solved for, and the length of time available for executing the GA. In general, a higher
**Mutation Rate** should be used where faster, less ideal solutions are desired, and a lower **Mutation Rate** should be used where slower, closer-to-ideal solutions are desired.

### 6.6.2 Crossover Rate

We have measured the effectiveness of **Crossover Rates** between 10% and 99%, with the results depicted in Figure 6.12. Note that we performed higher resolution tests above 90%, as **Crossover Rates** in the range of 90% to 99% are generally considered good default values for GAs[53]. As can be seen in Figure 6.12, **Crossover Rates** between 20% and 99% yielded similar results. This indicates that the selection of a **Crossover Rate** for our GA should be made based on how fast a solution is desired, with a higher **Crossover Rate** used to converge with fewer generations, and a lower **Crossover Rate** used to converge with more generations, but to a higher level of fitness.

### 6.6.3 Crossover Type

We have tested each of Random, Roulette and Tournament **Crossover Types**, with Single Point, Two Point, and Uniform selection mechanisms. The results are depicted in Figure 6.13. Figure 6.13 shows that the Tournament technique with a Two Point selection mechanism yields the greatest utility for our test systems, followed by the Tournament
technique with a Uniform selection mechanism.

### 6.6.4 Population Size

The time taken to run a GA for a fixed number of generations depends heavily upon the GA’s Population Size. Experimentation has shown that doubling the Population Size will roughly double the processing time required to reach a given number of generations. This meets expectations, since the fitness of each population member must be evaluated for each generation. However, more members are evaluated at each generation (yielding better results), so a balance must be struck. We have evaluated Population Sizes by testing the GA for 60 seconds with each Population Size, and recording the resulting utility, shown in Figure 6.14. Figure 6.14 shows that for our system, a Population Size of 40 yielded the highest average utility on our test problems. It should be noted, however, that this value will change depending on the capabilities of the system performing optimisation - a system with more processing power and larger or faster caches may benefit from a larger population.
Figure 6.13: Crossover Type

Figure 6.14: Population Size
6.6.5 Plateau Length

In order to determine how plateaus should be detected, the GA Solver was modified to output changes in fitness as they occur from generation to generation. The results are shown in Figure 6.15. Figure 6.15 shows the number of generations (x-axis) versus the utility (y-axis) for nine runs of the GA.

The mutation rate and crossover rates (shown on the legend as mutation rate (MR), crossover rate (CR) pairs) were varied in order to determine the effect, if any, of these on the shape of the resulting solution space. Although each result reached its highest value with a different number of generations, all reached a plateau at some point. In all cases, if after 6000 generations there was no improvement, there was no further improvement. Therefore, for this search space at least, the best plateau length is 6000. However, the ideal value for this parameter will depend not only on the system to be optimised, but also the value of optimisation to the service provider (i.e. if optimisation is more valuable, then a longer plateau length should be used in order to increase the quality and confidence in the optimisation result). Therefore, this value is more likely to be tuned based on the value of each system to be optimised and the capabilities of the optimisation system.

6.6.5.1 Genetic Memory Evaluation

Figure 6.16 shows the results of testing the GA Solver with genetic memory levels from 0% (no genetic memory) to 100% (every member of a new population is from genetic memory). The results give the average utility over 35 test systems (selected from all classes of system described in Section 6.7.2), each of which was mutated four times. Using genetic memory at any level (the values between 10 and 100 in Figure 6.16) provides better results than not using any genetic memory (the value at 0 in Figure 6.16). The genetic memory level that yielded the highest results was 70%, with an average increase in utility of approximately 10% over no memory.

Figure 6.17 shows the average utility level at each generation for the GA Solver with 0% and 70% memory levels. Figure 6.17 demonstrates the effectiveness of genetic memory in terms of both how fast an optimal result is found, and the level of that result. Using genetic memory of 70%, the average utility level for the first generation was approximately 31.5, versus an average of 23 for the same test systems without genetic memory (0%
Figure 6.15: GA Solution Space
Figure 6.16: GA Memory Size Evaluation

Figure 6.17: GA Memory Effectiveness Evaluation
memory level. Whilst the average utility for 0% memory slowly increases towards that for 70% memory, the level starts to plateau approximately 5% lower than the average utility level for 70% memory.

6.6.6 Summary

Based on our evaluation of each GA parameter, the following provides a set of good general parameters for experiments using the GA:

- A Mutation Rate of 10%;
- A Crossover Rate of 80%;
- A Crossover Type of Tournament Two Point;
- A Population Size of 40;
- A Plateau Length of 6000; and
- A Genetic Memory size of 70%;

Unless otherwise stated, these are the parameters that are used in the experiments in the following sections.

6.7 Heuristic Optimisation Evaluation

This section presents our experiments used to evaluate the GA Solver by measuring it against the Enumerative Solver for various problem sets.

6.7.1 Methodology

The benefits of design-time optimisation have been demonstrated in Chapter 5. Therefore, we have focused on the quality of the results returned by the GA Solver compared to the (guaranteed optimal) Enumerative Solver. Since the effectiveness of the GA Solver is highly dependent on the target system and the changes it undergoes, we have performed a large series of experimental simulations. As discussed in Section 6.7.2, these simulations allow us to cover a wide range of system parameters and change rates, and increase confidence in the breadth of our results’ applicability.
We present results of experimenting with the GA Solver for solution quality (Section 6.7.2), performance (Section 6.7.3), and scalability (Section 6.7.4).

6.7.2 GA Solver Solution Quality

To assess the quality of the monitoring configurations found by the GA Solver, we have used the Pure Random Generators and Trended Random Generators described in Chapter 5 to generate search spaces (i.e. systems to be optimised) that we optimised with both the Enumerative Solver and GA Solver.

6.7.2.1 Pure Random Experiments

Fifty search spaces were generated using the Pure Random Generators described in Chapter 5. The test cases ranged in size from 4,000 to 500,000 possible combinations with an average of 85,000 possible combinations.

In all cases, the GA Solver was able to find an optimal solution (a solution with the same utility as the solution found by the Enumerative Solver) in less than 30 seconds, whilst the time to find an optimal solution using the Enumerative Solver ranged from 3 seconds to 15 minutes. These results demonstrate that at these problem scales (search space sizes up to 500,000 possible combinations), the GA-based solutions are as good as the enumerative solutions, whilst being discovered in much shorter time frames.

6.7.2.2 Trended Random Experiments

We have used the set of 50 randomly generated search spaces described in Chapter 5 to test the GA Solver. For each system, the optimal result (from the Enumerative Solver), best result from the GA Solver, and result from max monitoring were recorded. In addition, the total number of possible combinations for each input set, number of optimal combinations, and percent of optimal combinations are reported. Since experiments in Sections 6.7.2 and 6.7.3 establish the times required to process a search space using each method, run-times have not been reported again.
Table 6.2: Trended Random Test Results

<table>
<thead>
<tr>
<th>Type</th>
<th>#Combos</th>
<th>#Optimal</th>
<th>%Optimal</th>
<th>#Utils</th>
<th>#MSI</th>
<th>#Overheads</th>
<th>Enum</th>
<th>GA</th>
<th>Max</th>
<th>%Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>9216</td>
<td>8</td>
<td>0.087</td>
<td>60</td>
<td>22</td>
<td>25</td>
<td>2.028</td>
<td>2.028</td>
<td>0.811</td>
<td>60</td>
</tr>
<tr>
<td>a</td>
<td>3072</td>
<td>72</td>
<td>2.344</td>
<td>60</td>
<td>4</td>
<td>8</td>
<td>1.513</td>
<td>1.513</td>
<td>1.513</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>41472</td>
<td>112</td>
<td>0.270</td>
<td>36</td>
<td>92</td>
<td>43</td>
<td>0.061</td>
<td>0.061</td>
<td>0.013</td>
<td>79</td>
</tr>
<tr>
<td>a</td>
<td>442368</td>
<td>48</td>
<td>0.011</td>
<td>66</td>
<td>170</td>
<td>58</td>
<td>0.617</td>
<td>0.617</td>
<td>0.178</td>
<td>71</td>
</tr>
<tr>
<td>a</td>
<td>1536</td>
<td>768</td>
<td>50.000</td>
<td>30</td>
<td>8</td>
<td>15</td>
<td>0.079</td>
<td>0.079</td>
<td>0.079</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>6144</td>
<td>24</td>
<td>0.391</td>
<td>78</td>
<td>18</td>
<td>8</td>
<td>1.097</td>
<td>1.097</td>
<td>0.528</td>
<td>52</td>
</tr>
<tr>
<td>b</td>
<td>64</td>
<td>35</td>
<td>54.688</td>
<td>12</td>
<td>16</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>96</td>
<td>2</td>
<td>2.083</td>
<td>18</td>
<td>0</td>
<td>6</td>
<td>0.753</td>
<td>0.753</td>
<td>0.743</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>1</td>
<td>33.333</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0.147</td>
<td>0.147</td>
<td>0.147</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>12</td>
<td>6</td>
<td>50.000</td>
<td>12</td>
<td>3</td>
<td>1</td>
<td>0.077</td>
<td>0.077</td>
<td>0.077</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>6</td>
<td>3</td>
<td>50.000</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>4608</td>
<td>32</td>
<td>0.694</td>
<td>48</td>
<td>83</td>
<td>26</td>
<td>1.348</td>
<td>1.348</td>
<td>1.266</td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>64</td>
<td>4</td>
<td>6.250</td>
<td>42</td>
<td>16</td>
<td>1</td>
<td>1.522</td>
<td>1.522</td>
<td>0.518</td>
<td>66</td>
</tr>
<tr>
<td>c</td>
<td>288</td>
<td>96</td>
<td>33.333</td>
<td>48</td>
<td>14</td>
<td>21</td>
<td>0.321</td>
<td>0.321</td>
<td>0.321</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>2048</td>
<td>1</td>
<td>0.049</td>
<td>114</td>
<td>70</td>
<td>23</td>
<td>0.653</td>
<td>0.653</td>
<td>0.083</td>
<td>87</td>
</tr>
<tr>
<td>c</td>
<td>9216</td>
<td>16</td>
<td>0.174</td>
<td>90</td>
<td>45</td>
<td>16</td>
<td>1.754</td>
<td>1.754</td>
<td>1.615</td>
<td>8</td>
</tr>
<tr>
<td>d</td>
<td>648</td>
<td>30</td>
<td>4.630</td>
<td>24</td>
<td>12</td>
<td>5</td>
<td>1.526</td>
<td>1.526</td>
<td>1.526</td>
<td>0</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Type</th>
<th>#Combos</th>
<th>#Optimal</th>
<th>%Optimal</th>
<th>#Util</th>
<th>#MSI</th>
<th>#Overheads</th>
<th>Enum</th>
<th>GA</th>
<th>Max</th>
<th>%Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>1728</td>
<td>108</td>
<td>6.250</td>
<td>72</td>
<td>41</td>
<td>17</td>
<td>0.134</td>
<td>0.134</td>
<td>0.134</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>18432</td>
<td>128</td>
<td>0.694</td>
<td>30</td>
<td>16</td>
<td>16</td>
<td>0.977</td>
<td>0.977</td>
<td>0.818</td>
<td>16</td>
</tr>
<tr>
<td>d</td>
<td>6912</td>
<td>96</td>
<td>1.389</td>
<td>54</td>
<td>10</td>
<td>11</td>
<td>0.715</td>
<td>0.715</td>
<td>0.715</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>55296</td>
<td>288</td>
<td>0.521</td>
<td>48</td>
<td>59</td>
<td>19</td>
<td>2.342</td>
<td>2.342</td>
<td>1.415</td>
<td>40</td>
</tr>
<tr>
<td>e</td>
<td>3072</td>
<td>96</td>
<td>3.125</td>
<td>220</td>
<td>52</td>
<td>23</td>
<td>0.563</td>
<td>0.563</td>
<td>0.417</td>
<td>26</td>
</tr>
<tr>
<td>e</td>
<td>9216</td>
<td>4</td>
<td>0.043</td>
<td>140</td>
<td>30</td>
<td>18</td>
<td>7.952</td>
<td>7.952</td>
<td>1.549</td>
<td>81</td>
</tr>
<tr>
<td>e</td>
<td>256</td>
<td>8</td>
<td>3.125</td>
<td>220</td>
<td>0</td>
<td>11</td>
<td>1.96</td>
<td>1.96</td>
<td>1.001</td>
<td>49</td>
</tr>
<tr>
<td>e</td>
<td>1152</td>
<td>24</td>
<td>2.083</td>
<td>120</td>
<td>6</td>
<td>5</td>
<td>1.152</td>
<td>1.152</td>
<td>1.152</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>36</td>
<td>2</td>
<td>5.556</td>
<td>60</td>
<td>5</td>
<td>3</td>
<td>5.991</td>
<td>5.991</td>
<td>5.991</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>1536</td>
<td>4</td>
<td>0.260</td>
<td>200</td>
<td>249</td>
<td>54</td>
<td>0.58</td>
<td>0.58</td>
<td>0.13</td>
<td>78</td>
</tr>
<tr>
<td>f</td>
<td>6912</td>
<td>36</td>
<td>0.521</td>
<td>60</td>
<td>201</td>
<td>24</td>
<td>0.453</td>
<td>0.453</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>f</td>
<td>384</td>
<td>8</td>
<td>2.083</td>
<td>120</td>
<td>16</td>
<td>11</td>
<td>1.584</td>
<td>1.584</td>
<td>1.197</td>
<td>24</td>
</tr>
<tr>
<td>f</td>
<td>432</td>
<td>6</td>
<td>1.389</td>
<td>160</td>
<td>159</td>
<td>38</td>
<td>0.295</td>
<td>0.295</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>f</td>
<td>1728</td>
<td>288</td>
<td>16.667</td>
<td>60</td>
<td>66</td>
<td>5</td>
<td>1.014</td>
<td>1.014</td>
<td>1.014</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>98304</td>
<td>4</td>
<td>0.004</td>
<td>260</td>
<td>267</td>
<td>36</td>
<td>3.573</td>
<td>3.573</td>
<td>0.264</td>
<td>93</td>
</tr>
<tr>
<td>g</td>
<td>6144</td>
<td>32</td>
<td>0.521</td>
<td>100</td>
<td>48</td>
<td>9</td>
<td>2.588</td>
<td>2.588</td>
<td>0.931</td>
<td>64</td>
</tr>
<tr>
<td>g</td>
<td>93312</td>
<td>24</td>
<td>0.026</td>
<td>180</td>
<td>203</td>
<td>27</td>
<td>1.704</td>
<td>1.704</td>
<td>0.133</td>
<td>92</td>
</tr>
<tr>
<td>g</td>
<td>62208</td>
<td>48</td>
<td>0.077</td>
<td>140</td>
<td>58</td>
<td>14</td>
<td>13.567</td>
<td>13.567</td>
<td>12.907</td>
<td>5</td>
</tr>
<tr>
<td>g</td>
<td>20736</td>
<td>2</td>
<td>0.010</td>
<td>300</td>
<td>95</td>
<td>17</td>
<td>4.567</td>
<td>4.567</td>
<td>1.664</td>
<td>64</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Type</th>
<th>#Combos</th>
<th>#Optimal</th>
<th>%Optimal</th>
<th>#Utils</th>
<th>#MSI</th>
<th>#Overheads</th>
<th>Enum</th>
<th>GA</th>
<th>Max</th>
<th>%Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>46656</td>
<td>72</td>
<td>0.154</td>
<td>160</td>
<td>71</td>
<td>11</td>
<td>9.702</td>
<td>9.702</td>
<td>5.94</td>
<td>39</td>
</tr>
<tr>
<td>h</td>
<td>3024</td>
<td>4</td>
<td>0.132</td>
<td>108</td>
<td>19</td>
<td>5</td>
<td>0.741</td>
<td>0.741</td>
<td>0.652</td>
<td>12</td>
</tr>
<tr>
<td>h</td>
<td>45000</td>
<td>75</td>
<td>0.167</td>
<td>150</td>
<td>10</td>
<td>3</td>
<td>5.163</td>
<td>5.163</td>
<td>5.163</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>121500</td>
<td>4500</td>
<td>3.704</td>
<td>30</td>
<td>3</td>
<td>0</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>72000</td>
<td>225</td>
<td>0.313</td>
<td>54</td>
<td>19</td>
<td>8</td>
<td>0.162</td>
<td>0.162</td>
<td>0.064</td>
<td>60</td>
</tr>
<tr>
<td>h</td>
<td>48384</td>
<td>864</td>
<td>1.786</td>
<td>60</td>
<td>10</td>
<td>1</td>
<td>0.143</td>
<td>0.143</td>
<td>0.101</td>
<td>29</td>
</tr>
<tr>
<td>i</td>
<td>13824</td>
<td>8</td>
<td>0.058</td>
<td>90</td>
<td>0</td>
<td>2</td>
<td>2.556</td>
<td>2.556</td>
<td>2.416</td>
<td>5</td>
</tr>
<tr>
<td>i</td>
<td>1296</td>
<td>9</td>
<td>0.694</td>
<td>15</td>
<td>0</td>
<td>5</td>
<td>0.14</td>
<td>0.14</td>
<td>0.018</td>
<td>87</td>
</tr>
<tr>
<td>i</td>
<td>3456</td>
<td>64</td>
<td>1.852</td>
<td>36</td>
<td>8</td>
<td>1</td>
<td>0.141</td>
<td>0.141</td>
<td>0.141</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>2304</td>
<td>864</td>
<td>37.500</td>
<td>45</td>
<td>14</td>
<td>4</td>
<td>1.527</td>
<td>1.527</td>
<td>1.527</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>34560</td>
<td>2</td>
<td>0.006</td>
<td>75</td>
<td>47</td>
<td>5</td>
<td>4.934</td>
<td>4.934</td>
<td>2.962</td>
<td>40</td>
</tr>
<tr>
<td>j</td>
<td>2048</td>
<td>8</td>
<td>0.391</td>
<td>39</td>
<td>74</td>
<td>5</td>
<td>2.102</td>
<td>2.102</td>
<td>2.04</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>16384</td>
<td>48</td>
<td>0.293</td>
<td>9</td>
<td>54</td>
<td>25</td>
<td>0.198</td>
<td>0.198</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>j</td>
<td>131072</td>
<td>16</td>
<td>0.012</td>
<td>21</td>
<td>110</td>
<td>25</td>
<td>1.262</td>
<td>1.262</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>j</td>
<td>262144</td>
<td>768</td>
<td>0.293</td>
<td>24</td>
<td>40</td>
<td>12</td>
<td>0.874</td>
<td>0.874</td>
<td>0.395</td>
<td>55</td>
</tr>
<tr>
<td>j</td>
<td>524288</td>
<td>64</td>
<td>0.012</td>
<td>33</td>
<td>30</td>
<td>18</td>
<td>1.751</td>
<td>1.751</td>
<td>0.599</td>
<td>66</td>
</tr>
</tbody>
</table>

AVG: 36
Table 6.2 summarises the results of testing using each of the 10 input parameter set classes described above.

For each class, at least five tests were run, and each test run is recorded as a row in Table 6.2. Each row shows the class of test **Type**, the number of possible configurations for that test **#Combos**, the number of optimal configurations for that test **#Optimal**, percent of combinations that were optimal **%Optimal**, number of **Utility Functions #Utils**, number of monitoring setting impacts **#MSI**, number of overhead costs **#Overheads**, the total utility from the **Enumerative Solver Enum**, total utility from the **GA Solver GA**, total utility from maximum monitoring **Max**, and the percentage increase in utility of the **GA Solver** over the **Max** solution **%inc**.

In each case, the **GA Solver** was run for 250 generations, with a population size of 20. In every case, the **GA Solver** produced an optimal result (a result yielding utility the same as the enumerative solution). This demonstrates that the **GA Solver** is a potential replacement for the **Enumerative Solver**. Furthermore, the optimal results had on average 36% higher utility than maximum monitoring. The major weakness remaining for the **GA Solver** is that, due to the nature of the **GA heuristic**, the results it provides are not guaranteed to be optimal, nor can their closeness to optimality be calculated.

The percentage of optimal results column highlights the target size within the search space that must be found by the **GA**. The percentage of optimal results was as low as 0.006, when just 2 results in 34,560 were optimal (the last test using class i). The **GA Solver** repeatedly found one of these two optimal solutions in 250 generations, with a population size of 20. Therefore, the **GA Solver** could have sampled a maximum of 250 × 20 = 5,000 combinations, approximately one seventh of the total possible 34,560 combinations. The fact that the **GA Solver** always found an optimal solution in this case (and similar cases) demonstrates that the **GA Solver** is not just randomly encountering optimal solutions, i.e. the **GA Solver** is more than just a random sampler. It should be noted that as the size of search spaces increase, the percent of solutions that are optimal will usually decrease. This is due to the fact that in search spaces with fewer requirements, there are a greater percent of optimal solutions in terms of meeting stated requirements, even if those

---

3A plateau length of 6000 and population size of 40 were deemed to not be required here, since the **GA Solver** effectively found optimal solutions in every case using these lower values.
solutions provider a lower QoS (i.e., there are lots of ties). As the size and complexity of a search space grows, we expect that there will be relatively fewer ties, and therefore a lower percent of optimal solutions. This is supported by Table 6.2, which shows a trend of a lower percent of optimal solutions with increasing search space sizes.

6.7.3 Performance

To test the computational performance of the GA Solver, we executed both the GA Solver and Enumerative Solver on a series of randomly generated systems, with between 1 and 17 monitors (with corresponding search space sizes from 50 to $2 \times 10^6$). Figure 6.18 shows the computation time achieved for each technique.

Figure 6.18 shows the computation time in seconds required to find and apply a solution by the GA Solver and Enumerative Solver. Note that both axes are log scale. The GA Solver was faster than the Enumerative Solver for all search space sizes over 100 (representing 1-2 monitors). For these tests, the GA Solver achieved the same quality of output as the Enumerative Solver (i.e., all of the GA Solver’s results were optimal). As the size of the search space grew, the Enumerative Solver took up to 15 minutes to discover a solution, versus a maximum of 2 seconds for the GA Solver. Whilst the Enumerative Solver is efficient enough to solve small search problems, Figure 6.18 shows that the technique will not scale to larger problems. This is due to the size of the search space, which is exponential on the number of possible monitoring settings (number of parameters). The growth of the GA Solver, whilst superlinear, is close enough to linear that it can still be used to solve large problems, as discussed in Section 6.7.4.

6.7.4 Scalability

Although the GA Solver is more efficient than the Enumerative Solver, there will be some size of search space that is too large for the GA Solver to solve in a reasonable time (one hour) in order for the GA Solver to be effective for run-time optimisation. We have performed experiments to discover the scalability characteristics of the GA Solver, and the size of search space where default, maximum monitoring becomes more effective than the GA Solver. These experiments used increasingly larger systems, in order to discover the crossover point where the GA Solver could not reliably find a better than
Figure 6.18: Optimisation Techniques Computation Time
default (maximum) solution in under one hour. The size of search space where the GA Solver yielded a lower result than max monitoring is approximately $10^{160}$. Note that this will vary from case-to-case, as variables such as the amount of overlapping monitoring, amount of cost functions, and amount of utility functions contribute to the processing time for the GA Solver.

$10^{160}$ combinations will allow for 330 Monitoring Capabilities with an average of three Monitoring Levels each. This would allow for 5 Monitors observing 5 Quality Types of 13 Services at 3 Monitoring Levels. However, the amount of Services, Monitors, and Quality Types can be increased since not all Monitors will be observing all Quality Types and Services. If we assume that a Monitor has a 30% probability of monitoring any specific Quality Type, and that there is an average of 100% overlap from Monitors to Services (i.e. on average, each Service is monitored by 2 Monitors), the amount of potential Monitoring Capabilities could be expanded to up to $330 \times (100/30) \times (100/10) = 11,000$ potential Monitoring Capabilities. This would allow for 20 Monitors to observe 5 Quality Types of 100 Services, assuming that those 20 Monitors each observed an average of 100/10 = 10 Services and $5 \times .33 = 1.65$ Quality Types.

This figure is further improved through search space pruning. For a large test system, a search space of size $5 \times 10^{253}$ was pruned to $6 \times 10^{161}$. If then, we take the figure of $10^{250}$ as the bound for the number of Monitoring Configurations that can be handled with the GA, a system with 530 Monitoring Capabilities and average of 3 Monitoring Levels could be solved. This would allow for approximately twice as many Services in the above example (i.e. 20 Monitors, 5 Quality Types, 200 Services) if the probabilities of monitoring any given Quality Type are still 30% and there is 100% overlap.

This figure of 530 Monitoring Capabilities, and 17,000 potential Monitoring Configurations will be taken as the limit for the GA, in a run-time system.

The limitations above are based on our own systems used for the experiments above, and will change depending on system requirements such as acceptable optimisation time frames and processing capabilities available. Figure 6.19 shows the results of an experiment we performed to determine the general trend of performance for the GA Solver. Figure 6.19 shows the time taken for the GA Solver to find a solution using a plateau
length of 1,000. In Figure 6.19, the horizontal axis shows the number of search space parameters (i.e. number of possible monitoring settings - the width of the search space), and the vertical axis shows the GA Solver’s computation time divided by that width. This allows for a direct comparison of the effect of increasing search space size on the computation time of the GA Solver. Were the trend-line in Figure 6.19 horizontal, then the GA Solver would scale linearly. However, the trend-line has a positive gradient (0.0002), which indicates that the GA Solver is slightly super-linear. This means that, if a search space size is doubled, then the time to solve that search space would be approximately 2.0002 times longer.
6.8 Effects of Run-Time Optimisation

We have performed a series of simulations to determine the effect that re-optimising at run-time has on resulting QoS and net utility. We have generated and optimised 10 baseline systems, one for each class of system in Table 5.3. For each system, we pre-generated a series of 5 changes to monitoring and QoS values, monitoring costs, system capabilities (adding and removing monitors and services), and system load levels.

To get a baseline, each system was optimised, run for 6,000 seconds, halted and modified, and run for another 6,000 seconds, for a total of 4 modifications and 5 runs per system. For these baseline cases, re-optimisation was not performed between system modifications.

To measure the effectiveness of run-time optimisation, the above tests were repeated, this time with re-optimisation occurring after each system modification. Figure 6.20 shows the results of these experiments. In Figure 6.20, the ”Re-optimisation Utility Increase” is calculated as the difference between the utility provided by the original, design-time optimisation (performed on the first system generation) and the utility provided by the new, run-time optimisation (performed for each new system generation). Figure 6.20 shows the increase in utility for each system class a to j as well as the average over all classes.

When performing run-time reoptimisation, the average increase in utility across all 10 classes of system (when compared to an initial optimisation that occurred before any system modifications) for generations one to four was approximately 14%, 19%, 26%, and 28%, respectively. The average across all four generations was approximately 20%. It should be noted that the increase in utility for a particular system may not always increase, as is demonstrated by the results for most classes in Figure 6.20. This is because any system modifications after the first may (by chance) cause the original, design-time optimisation to become optimal or close to optimal again. However, on average utility continued to increase as more system modifications were performed.

6.9 Summary

In this chapter, we have presented a framework and technique for run-time management and optimisation of a WS monitoring system. This framework provides the capability
Figure 6.20: Effect of Continuous Run-time Re-optimisation
to continuously monitor a WS monitoring system at run-time, and perform optimisation when changes occur. We have also presented a heuristic optimisation routine that has been used to efficiently solve the run-time optimisation problem. We have demonstrated that the use of GA-based optimisation with search space pruning allows for systems with up to $10^{160}$ possible monitoring configurations to be optimised within one hour, compared to the enumerative approach which will only solve for systems with up to approximately $10^6$ possible monitoring configurations within this time-frame. Furthermore, the GA-based solver found optimal solutions for all experiments that could be solved by the enumerative technique. The run-time optimisation system has been implemented and tested, with an average of 20% increase in utility achieved across a wide range of randomly generated target systems, when run-time optimisation is enabled.
Chapter 7

Automated Monitoring Impact Analysis

We demonstrated the need for the automated control of monitoring configurations at run-time in Chapter 6. Our technique relied on the impacts of monitors being determined manually through testing of the system at design-time. However, monitoring impacts cannot always be established before run-time, since monitors may be bound to or re-configured at run-time, and their host systems in a Web Service environment may be reconfigured after deployment. Furthermore, monitoring impacts themselves may change at run-time as the monitors or the Web Service system are reconfigured. Knowledge of monitoring impacts is required to determine the business cost of monitoring efforts, optimise a Web Service system’s Quality of Service (QoS) at run-time, or balance system load. The accuracy of this monitoring impact knowledge is important, since any error may lead to poor choices in monitoring or system configurations, and ultimately in either lower QoS (if monitoring impacts are underestimated) or lower monitoring coverage (if monitoring impacts are overestimated).

In this chapter, we introduce a method that can determine the impacts of monitors on delivered QoS at run-time. By analysing run-time monitoring logs, this method accurately determines monitoring impacts as the system changes, whilst handling external noise from un-modelled factors.

Section 7.1 describes the Web Service monitoring impact analysis problem. Section
7.2 describes our analysis technique. Section 7.3 describes experiments used to validate and measure the effectiveness of our analysis technique.

7.1 Analysis of Monitoring Impacts

In order for optimisation to be applied to a suite of Web Service monitors, the impacts of each of those monitors must be accurately determined. We previously measured monitoring impacts by manually testing each monitor and Web Service at deployment time. In addition to requiring human intervention, this manual analysis becomes invalid if the system changes at run-time (e.g. if a monitor is assigned more or less resources, relocated or modified). Therefore, we require a method to automatically measure the impact of Web Service monitors at run-time. This analysis should provide accurate measurements of the overhead and monitoring setting impacts of monitors in the Web Service provider’s system. It should not modify the monitoring system (e.g. manually re-configuring monitors). In this section, we define the monitoring impact analysis problem.

Knowledge of monitoring impacts may come from various sources. These sources include developer’s design-time measurements, manual or automatic deployment-time testing, and manual or automatic run-time testing. Design or development-time testing may not yield accurate results, since the testing is not being performed on the production system. Deployment-time evaluation methods may yield accurate results, however that accuracy may decrease over time as the run-time system is modified. It is therefore beneficial if the performance impact of monitors can be measured on the run-time system, and updated at run-time as the system changes. This may be achieved through either experiments on the run-time system, or analysis of a system’s logs. Whilst experiments on a run-time system allow for a greater level of control, they require the run-time system to be available for modification (enabling and disabling of monitors).

7.1.1 Scenario

Figure 7.1 shows an example monitoring scenario. In this figure, book_flight represents a service offered by iTavel to public consumers. Consumers access the book_flight service through the book_flight Proxy. iTavel may monitor service invocations with either
of the **Interceptor** or **Eavesdropper** monitors. Both monitors measure the **response time** and **reliability** of the **book_flight** service, but use different methods to do so. The **Interceptor** monitor acts as a proxy, redirecting a message and analysing it en-route to the service or consumer. The **Eavesdropper** monitor passively captures network-level packets for analysis. Each of the monitors can be set to measure none, one, or both quality types.

In this system, the problem is to determine the impacts of the two monitors on the **response time** and **reliability** of the **book_flight** service. Whilst this is a simple scenario, it allows us to effectively demonstrate the impact analysis problem described below. A case study analysis in Chapter 8 provides a more complex scenario for the impact analysis technique.

### 7.1.2 Problem Definition

The monitoring impact analysis problem is to determine the impacts that each monitor has on each QoS being delivered by a Web Service provider. Any monitor in the system may impact any QoS, regardless of whether that monitor observes that QoS or not. This is due to the monitors and services sharing the same, finite set of resources. The impacts from each monitor will vary according to the actions being performed by that monitor. In the simplest case, reducing the sampling rate of a monitor will reduce that monitor’s impact level. In addition, if a monitor is capable of measuring multiple quality types, then the impact of that monitor may vary depending on the quality types measured. Therefore, the impact analysis problem is to determine all of the QoS impacts of each possible monitoring

---

Figure 7.1: iTravel Scenario
action in the system. It should be noted that external noise is a significant contributing factor to QoS levels in Web Service systems. As such, any impact analysis method should be able to function under various levels of noise.

Recall that monitoring impacts are the reductions in QoS that occur when monitors are used. For example, using an intercepting monitor to measure response time of a service may increase the response time of that service. We model monitoring with two separate impacts: monitoring overhead, and monitoring setting impacts. We introduced and formally defined the sets of monitoring setting impacts and monitoring overhead impacts in Chapter 5. A summary of all formal definitions is provided in Appendix B.

When a monitoring configuration \( mc \in MCON \) is used, the set of impacts of that configuration \( IM \) reduce the ideal quality level \( IQL \) of some services and quality types, as defined in Chapter 5. The recorded impact \( (ri) \) on a quality \( q \) of service \( s \) is the difference between the ideal quality level \( IQL(s, q) \) and measured quality level \( MQL(s, q) \) for that quality \( q \) and service \( s \). The goal of monitoring impact analysis is to analyse records that provide sets of monitoring configurations with their associated recorded impacts \( (ri) \), and determine which monitors are responsible for which impacts in the system, and whether those impacts are overhead or monitoring setting impacts.

### 7.2 Impact Analysis Technique

Since manual control of monitors for monitoring impact measurements may not always be available and cannot be used for historical analysis, we have developed a method for determining the impact of individual monitors by analysing run-time logs from a suite of monitors. To determine the impacts that each monitor in the system has on each QoS, we analyse logs of delivered QoS levels with their corresponding monitoring configurations (where a monitoring configuration describes the configuration settings applied to every monitor in the system). We calculate the impact of individual monitoring settings within those configurations by comparing different monitoring configurations and their delivered QoS levels. This technique allows for the complete set of monitoring impacts to be determined from monitoring records without any initial monitoring impact knowledge. However, if monitoring records are not diverse enough (with a wide range of monitoring configurations), then monitoring configurations may have to be manually selected that
provide information required for impact analysis.

Our impact analysis technique is designed to be applied on a Web Service provider’s system with multiple services and monitors, and multiple quality types being monitored and impacted. We assume that logs of both delivered QoS and monitoring configurations are available from normal operation, which allows us to measure monitoring impacts without actively executing (probing) services. We also assume that the impacts of monitoring are costly enough that a Web Service provider would not simply enable all monitors all of the time.

The impact analysis technique derives individual monitoring impacts from a set of monitoring records (logs). Monitoring records consist of a monitoring configuration (stating what monitors were enabled, what QoSs they were monitoring, and at what levels) and a set of measured QoS levels. Note that a measured QoS level is an aggregate, observed QoS that embodies the impacts of all monitors. Our impact analysis technique uses these measured QoS levels to determine monitoring impacts of individual monitors and monitoring settings.

Figure 7.2 shows a small example set of monitoring data. In this figure, the measured response time is reported for four different monitoring configurations. There are two monitors (M1 and M2), and monitor M1 can measure both response time (RT) and reliability (RL), whilst monitor M2 can only measure response time. The baseline
response time for this example is 0.5 seconds, shown with a horizontal, dotted line. The reported response times are all above this level, meaning that the monitors have increased the response time of the service under evaluation. By analysing the response time for each monitoring configuration, we can see that: Monitor M2 has no impact (compare the first and fourth configurations, which have the same response time but M2 enabled and disabled, respectively); the monitoring setting impact of using monitor M1 to measure response time is 0.5 seconds (compare the second and third configurations); the monitoring setting impact of using monitor M1 to measure reliability is 1 second (compare the first and second configurations); and the overhead impact of monitor M1 is 0.5 seconds (using the second configuration, we know that the net per-setting impacts of monitor M1 are 0.5s and 1s, giving a difference of 2.5-1.5=1s, which is 0.5 seconds higher than baseline). Whilst there is no noise in this example, it illustrates the general technique we use to measure monitoring impacts.

In general, we first search for a monitoring record or set of records where there is no impact on a QoS, and assign any enabled monitors as having no impact on that QoS. Then, we search for monitoring records with impacts on that QoS and only one unknown monitoring impact component (e.g. we might have a monitoring record with only one monitoring action having taken place), and we assign all un-accounted for monitoring impacts to this monitoring action.

The Impact Analyser takes as input all relevant QoS records along with their associated monitoring configurations. Using a set of known baseline QoS levels, the impact of each monitoring configuration (as a percentage) on each Quality/Service pair (QoS) is calculated. The Impact Analyser calculates both the overhead impacts for each monitor and monitoring setting impacts for each monitoring setting’s monitoring level onto each QoS. Each target QoS is considered independently. For example, the set of monitoring impacts on response time of the book_flight service may be determined independently from the set of monitoring impacts on reliability of the book_flight service (even though there may be a correlation between these QoS types).

Our impact analysis technique involves a record pre-processing stage and an impact analysis stage involving three sub-stages, described in the following sections.
Table 7.1: Raw QoS Records

<table>
<thead>
<tr>
<th>Monitoring Configuration</th>
<th>QoS Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_e Q_{rt} S_b$</td>
<td>$M_i Q_{rl} S_b$</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>1 1 0 1</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>1 1 0 1</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>1 1 0 1</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>1 1 0 1</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

7.2.1 QoS Record Pre-processing

The first stage for impact analysis is to pre-process QoS records. Our QoS records consist of pairs of monitoring configuration and delivered QoS. Delivered QoS levels are defined as percentages of ideal QoS. For example, a delivered QoS of 90% for response time of book_flight means that there has been a 10% impact on response time of book_flight (ignoring noise). The monitoring configuration (defined earlier) gives the monitoring level (e.g., sampling rate) that each monitoring capability was set to. The delivered QoS, $DQ = \{(s, q) \mapsto mql|\forall s \in S, q \in Q\}$ gives all monitored QoS levels $mql$ under that monitoring configuration. The raw QoS records form the multiset $RQ = \{mc \mapsto DQ\}$. Table 7.1 shows an example multiset of raw QoS records for the iTravel scenario, where $M_e$ is the Eavesdropper monitor, $M_i$ is the Interceptor monitor, $S_b$ is the book_flight service, $Q_{rt}$ is response time, and $Q_{rl}$ is reliability. For simplicity, we have only two monitoring levels (0 and 1), representing a monitor being disabled or enabled, respectively.

Table 7.1 shows some sample iTravel QoS records with three different monitoring con-
figurations. In Table 7.1, each row represents one monitoring record (i.e. the result of monitoring a single service execution under a particular monitoring configuration). The values in the Monitoring Configuration columns (the first four columns) represent the monitoring level associated with each monitoring capability, where $M_eQ_{rt}S_b$ represents the Eavesdropper monitor observing response time of book_flight, $M_eQ_{rt}S_b$ represents the Eavesdropper monitor observing the reliability of book_flight, $M_iQ_{rt}S_b$ represents the Interceptor monitor observing the response time of book_flight, and $M_iQ_{rt}S_b$ represents the Interceptor monitor observing reliability of book_flight. For example, monitoring configuration (1,0,0,0) in the first four rows describes a record where the Eavesdropper monitors book_flight’s response time at monitoring level 1 (completely enabled), and each other monitoring capability was set to monitoring level 0 (completely disabled). There are four records in Table 7.1 with this configuration, meaning that whilst this configuration was active, the book_flight service was executed four times. The measured QoS levels are shown in the two right hand columns of Table 7.1. The response time was measured as 85, 83, 87, and 85 percent of book_flight’s ideal response time. Reliability was not measured, as the configuration does not include a monitor that measures it.

Each monitoring configuration in Table 7.1 has four results associated with it. The record pre-processing component of the IMPACT ANALYSER averages these results and uses this set of averaged results to calculate the impact levels of each monitoring configuration. The impact levels for a particular record are the ideal QoS levels minus the delivered QoS levels. For clarity, we have assumed all ideal QoS levels are 100%. However, this ideal QoS level may be modified to account for factors such as system load by making ideal QoS a function of load. Table 7.2 shows an example output from the QoS RECORD PRE-PROCESSOR (a set of processed, averaged QoS records). The averaged results for the three monitoring configurations from Table 7.1 are highlighted in bold in Table 7.2. The processed QoS records are defined as the set $PQ = \{mc \mapsto AI\}$, where the average impact $AI(s,q) = IQL(s,q) - AQ(s,q) \forall s \in S, q \in Q$, and the average quality set $AQ = \{(s,q) \mapsto aql|\forall s \in S, q \in Q\}$ and the average quality level $aql$ for a given monitoring configuration $mc$ is $aql = \text{average}(RQ(mc)(s,q))$. 

212
Table 7.2: Processed QoS Records

<table>
<thead>
<tr>
<th>Monitoring Configuration</th>
<th>Impact Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_eQ_{rt}S_b$</td>
<td>$M_eQ_{rt}S_b$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

7.2.2 Impact Analysis

Impact analysis is performed after QoS records have been pre-processed. Table 7.2 shows an example set of processed QoS records. These are used by the IMPACT ANALYSER to calculate the set of QoS impacts of each monitor. In contrast to the set of raw QoS records in Table 7.1, the processed QoS records in Table 7.2 contain only one row per monitoring configuration, with QoS levels given as the average QoS impacts for that configuration (defined above).

After record processing, the IMPACT ANALYSER calculates the monitoring setting impacts and overhead impacts that each monitor in the system has on each QoS. Impact analysis is carried out in three steps: null impact analysis, monitoring setting impact analysis, and overhead impact analysis.

7.2.2.1 Null Impact

At the Null Impact Analysis stage, for each QoS that may be impacted (the target QoS), we determine which monitors and monitoring settings have no impact on that target QoS.
This is achieved by searching for monitoring records where the target QoS is not impacted at all and assigning null impacts to all monitors and monitoring settings used when that record was generated. Null impact analysis requires user-supplied cutoffs for mean and variance levels. The target mean cutoff gives the mean value for which an impact level is assumed to be zero (e.g. a value of 0.01 may be assumed to be 0), and the variance level is the maximum acceptable variance from the sample set. Whilst any confidence measurement can be used, for simplicity we have used the variance of the set of raw QoS records (where variance $v$ of set $X$ is defined as $\text{mean}(X^2) - \text{mean}(X)^2$). This provides a simple measure of the variability of the monitoring records.

Algorithm 5 Null Impact Analysis

1. $\text{var\_cutoff} := \text{[user input]}$
2. $\text{mean\_cutoff} := \text{[user input]}$
3. for all $s$ in Services do
   4. for all $q$ in Quality\_Types do
      5. for all $qr$ in $PQ$ do
         6. $v := \text{variance}(\text{RQ}(\text{dom}(qr)))(s,q)$
         7. if $v \leq \text{var\_cutoff}$ and $\text{ran}(qr)(s,q) \leq \text{mean\_cutoff}$ then
            8. for all $ms \in \text{dom}(qr), ms.ml > 0$ do
               9. $\text{ms\_kn}\{ms\} = \{ms.ml \mapsto 0\}$
              10. $\text{oh\_kn}\{ms\text{.monitor}\} = 0$
            end if
         end for
      end for
   end for
end for

Algorithm 5 describes null impact analysis. We enumerate the set of all QoSs that may be impacted (Lines 3-4), for all processed QoS records $PQ$ (Line 5), and determining which monitoring settings and overheads can be safely assigned with no impact on any particular QoS. That is, if there exists a QoS record $qr \in PQ$ with some monitors enabled $ms \in \text{dom}(qr), ms.ml > 0$ (Line 8), and the QoS record states that there is no impact for a particular monitored QoS $\text{ran}(qr)(s,q)$ within a given tolerance (Line 7), then all enabled
monitors in that QoS record may be assumed to have no monitoring setting or overhead impact on that QoS. When these records are found, the set of null monitoring setting impacts is recorded in the $ms_{kn}$ array (line 9), and the set of null overhead impacts is recorded in the $oh_{kn}$ array (Line 10).

For example, the fourth row in Table 7.2 with monitoring configuration (1,1,0,0) has a recorded 0% impact level on the reliability of book_flight. The variance of the raw QoS records (not shown) producing this value was 0.05. Since the mean is below a target value of 0.1 and the variance is below a target value of 0.1, we may assume that the overhead of the Eavesdropper monitor on reliability of book_flight is 0% (i.e., no impact), and that the monitoring setting impacts of using the Eavesdropper to observe response time and reliability of book_flight are also 0%. Conversely, the last row in Table 7.2 has an impact level of 2% on reliability of book_flight, and so this record cannot be used to assign null impacts to any of the monitors used. Additionally, the last record (1,1,1,1) cannot be used because of both the impact levels being above 0.1 and the variance of the raw monitoring records being above 0.1.

Ideal variance and mean impact cutoff (var_cutoff and mean_cutoff in Algorithm 5) will depend on the target system. In general, if there is a larger dataset available, the variance cutoff can be decreased and results will still be obtained, since there will be enough results left for processing after pruning. However, if there is only a small dataset available, the system designer will have to trade off between false negatives (not assigning null impact to monitors without impact) and false positives (assigning null impact to monitors which have impact).

### 7.2.2.2 Monitoring Setting Impact

After null impact analysis, the processed QoS records are used to determine monitoring setting impacts. This is achieved by searching for ‘mirrored’ monitoring records, which are pairs of records with only one difference in their configuration settings, e.g. (1,1,1,1) and (1,1,0,1). Any difference in impact between these records can then be assigned to the mirrored monitoring setting, in this case the third monitoring setting in the QoS record. Using the data from Table 7.2 as an example, we may wish to find the impact of the Eavesdropper monitor observing the response time of book_flight ($M_r Q_{rt} S_b$) on the
response time of book_flight \((Q_{rt}S_b)\). Monitoring configuration \((1,1,1,1)\) gives a total impact of 28%, whilst the mirrored configuration \((0,1,1,1)\) gives a total impact of 23%. The difference between these records \((5\%)\) can then be assigned to the Eavesdropper monitor observing response time of book_flight \((M_eQ_{rt}S_b)\).

Algorithm 6 shows our monitoring setting impact analysis algorithm. We enumerate the set of all services \(s\) (Line 1) (from Table 7.2 this would be book_flight) and quality types \(q\) (Line 2) (from Table 7.2 this would be response time and reliability). For each QoS, we search through all monitoring settings \(ms_i\) that may impact them (Line 3) (from Table 7.2 this would be \(M_eQ_{rt}S_b\), \(M_eQ_{rl}S_b\), \(M_iQ_{rt}S_b\), and \(M_iQ_{rl}S_b\)). For each monitoring setting \(ms \in M^3\) and QoS \((s,q)\), we attempt to calculate the impact of that monitoring setting on that QoS, based on the processed monitoring records \(PQ\). This is achieved by searching the set \(PQ\) (Line 5) (This corresponds to Table 7.2) for records where the target monitoring setting \(ms_i\) is used with a monitoring level greater than 0 \(\exists qr(ms_i)\) and \(\text{dom}(qr)(ms_i) > 0\) (Line 6). That is, we search for for a particular monitoring record \(pq\) where the monitoring setting \(ms_i\) is used at a monitoring level greater than 0. For each matching record, we search for a mirrored record \(mc_{\text{mirrored}}\) in \(PQ\) (Line 8) and ensure that for that record, the QoS of interest was monitored. A mirrored record is defined as a processed monitoring record that has the same monitoring configuration with the exception of the monitoring setting of interest, which has a monitoring level of 0 (Lines 20 to 24). As mentioned above, the difference between these records will be the impact of the monitoring setting (at the monitoring level) of interest. Once we have searched all processed records for a given monitoring level, we average the impacts from each mirrored pair and add this value to our hash table of monitoring setting impact analysis knowledge \(ms_{kn}\) (Line 14).

For example, consider the impact on response time of book_flight from the Interceptor monitor observing reliability of book_flight. In Algorithm 6, this would correspond to \(s\) being equal to book_flight (Line 1), \(q\) being equal to response time (Line 2), and \(ms_i\) being equal to (Interceptor, book_flight, reliability) \(\mapsto 1\). Lines 5-6 search for each processed quality record with this monitoring setting in use (rows 6,7,8,9,10 and 12 in Table 7.2). Lines 7-8 search for a mirrored monitoring configuration. For example,
the monitoring configuration in row 6 of Table 7.2 is mirrored with that in row 1, and
the monitoring configuration in row 12 is mirrored with that in row 11. However, we also
require that the mirrored monitoring configuration have a measured impact level on the
current QoS (\texttt{defined(PQ(mc\_mirrored)(s,q))} at Line 8). For this example, the config-
uration at row 0 in Table 7.2 does not meet this requirement (as response time is not
measured in this case), however the configuration at row 11 does. The impact delta (Line 9)
between rows 12 and 11 for this QoS is 28\% (row 12) - 25\% (row 11) = 3\%. Similarly,
the impact delta between rows 10 and 4 is (24\% - 20\% = 4\%). Finally, rows 7 and 5
give a delta of (8\% - 4\% = 4\%). Given these three results, the average impact (3.7\%) on
response time of book\_flight from monitoring reliability of book\_flight using the
Interceptor monitor would be added to the ms\_kn hash (Line 14).

7.2.2.3 Monitoring Overhead Impact

Once all monitoring setting impacts are known, the monitoring overhead costs can be
determined. For Overhead Impact Analysis, we assign any remaining, unaccounted for
impacts to monitoring overheads. This is achieved in a similar way to monitoring setting
impact analysis, and involves searching for pairs of monitoring records with a monitor
enabled and disabled, and all other impacts either known or unchanged. The overheads
for each monitor are the remaining, unaccounted for impacts from each QoS Record. For
example, if we have already determined each monitoring setting impact from Table 7.2,
and know that $M_eQ_{rt}S_b$ has a 5\% impact on $Q_{rt}S_b$ and $M_eQ_{rl}S_b$ has a 5\% impact on $Q_{rt}S_b$,
then the monitoring configuration $(1,1,0,0)$ will show that the remaining unaccounted for
impact is $(20 - 10 = 10\%)$, the overhead impact of $M_e$.

Algorithm 7.2.3 shows our monitoring overhead impact analysis algorithm. We enu-
merate the set of all services $s$ (Line 2) and quality types $q$ (Line 3) that may be impacted,
for all monitors $m$ that may impact them (Line 4). For each monitor $m \in \texttt{Monitors}$ and
QoS that it may impact, we enumerate the set of processed monitoring records $PQ$ (Line
6) (corresponding to Table 7.2). We restrict this set to those records with monitor $m$ en-
abled ($m \in \texttt{monsInUse(qr)}$) (Line 7) and ensure that the QoS of interest was monitored
(\texttt{defined(PQ(qr)(s,q))}) (Line 7). The \texttt{monsInUse} subroutine (Lines 29-38) enumerates
all monitoring settings $ms$ in the monitoring configuration $\texttt{dom(qr)}$ of a quality record $qr
Algorithm 6 Monitoring Setting Impact Analysis

1: for all s in Services do
2:     for all q in Quality.Types do
3:         for all ms_i in MS do
4:             deltas[] := ∅
5:                 for all qr in PQ do
6:                     if exists(qr(ms_i)) and dom(qr)(ms_i) > 0 then
7:                         mc_mirrorred := mirrored(dom(qr), ms_i)
8:                         if exists((PQ(mc_mirrorred)) and defined(PQ(mc_mirrorred)(s,q))) then
9:                             delta := (PQ(qr))(s,q) - (PQ(mc_mirrorred))(s,q)
10:                            push(deltas[], delta)
11:                     end if
12:                 end if
13:         if deltas[] != ∅ then
14:             ms_kn{ms_i \mapsto (s,q)} := average(deltas[])
15:         end if
16:     end for
17: end for
18: end for
19: end for
20: sub mirrored(mc: Monitoring Configuration, ms_i: Monitoring Setting
21:     mc_mirror := mc
22:     mc_mirror(ms_i) := 0
23: return mc_mirror
24: end sub
(Line 31) that have a monitoring setting greater than 0 (Line 32) (i.e., are enabled), and adds the monitor \((\text{dom}(ms).m)\) for that record to the set of monitors in use \(mi\) (Line 34) if it is not already there. For each of these records with the monitor of interest enabled, we proceed if all monitoring setting costs are known \((\text{defined}(ms.kn{ms_j \mapsto (s,q)}))\) (Line 8) and all other monitoring overhead costs \((\text{defined}(oh.kn{mi \mapsto (s,q)}))\) (Line 9) are known. For each of these records, the sum of known monitoring setting impacts from \(ms.kn\) for all enabled monitoring settings \(ms \in \text{dom}(qr)\) (Lines 11-13) and monitoring overhead impacts from \(oh.kn\) for all other monitors in use \(mi \in \text{monsInUse}(qr)\) (Lines 14-16) is subtracted from the recorded impact (Line 17), giving the as-yet unaccounted for impact in that record, belonging to the overhead of monitor \(m\), which is added to an array of \(\text{deltas}\) (Line 18). The average of all of these impact deltas is calculated and recorded in \(oh.kn\), a hash table for monitoring overhead impact knowledge (Line 24).

**Algorithm 7.2.3 Monitoring Overhead Impact Analysis**

1. \(ms.kn := [\text{Monitoring Setting Impact Analysis output}]\)
2. \(\text{for all } s \text{ in Services do}\)
3. \(\text{for all } q \text{ in Quality Types do}\)
4. \(\text{for all } m \text{ in Monitors do}\)
5. \(\text{deltas[]} := \emptyset\)
6. \(\text{for all } qr \text{ in PQ do}\)
7. \(\text{if } m \text{ in monsInUse}(qr) \text{ and defined(PQ(qr)(s,q)) then}\)
8. \(\text{if } (\text{defined}(ms.kn{ms_j \mapsto (s,q)}) \text{ for all } ms.j \text{ in } \text{dom}(qr)) \text{ then}\)
9. \(\text{if } (\text{defined}(oh.kn{mi \mapsto (s,q)}) \text{ for all } mi \text{ in } \text{monsInUse}(qr), mi != m) \text{ then}\)
10. \(\text{impacts} := 0\)
11. \(\text{for all } ms \text{ in } \text{dom}(qr) \text{ do}\)
12. \(\text{impacts} += ms.kn{ms \mapsto (s,q)}\)
13. \(\text{end for}\)
14. \(\text{for all } mi \text{ in } \text{monsInUse}(qr), mi != m \text{ do}\)
15. \(\text{impacts} += oh.kn{mi \mapsto (s,q)}\)
16. \(\text{end for}\)
17. \(\text{delta} := PQ(qr)(s,q) - \text{impacts}\)
push(deltas[], delta)
end if
end if
end if
end for
if deltas != ∅ then
oh_{kn}(m \mapsto (s,q)) := average(deltas[])
end if
end for
end for
end for
sub monsInUse(qr: Processed Quality Record)
mi[] := ∅
for all ms in dom(qr) do
if ms > 0 then
if !defined(mi[dom(ms).m]) then
push(mi[], dom(ms).m)
end if
end if
end for
end sub

As an example for Algorithm 7.2.3, consider the overhead impact of the Interceptor monitor on response time of the book_flight service. We are assuming that monitoring setting impact analysis has been performed, and the monitoring setting impacts on response time of book_flight are 5% for the Eavesdropper observing response time, 5% for the Eavesdropper observing reliability, 4% for the Interceptor observing response time and 4% for the Interceptor observing reliability. We assume that the overhead of both monitors is not known. In this case, the value for s is book_flight (Line 2), the value for q is response time (Line 3) and the value for m is Interceptor (Line 4). Line 6 enumerates all processed monitoring QoS records in Table 7.2. Line 7 then restricts this set to those monitoring records in Table 7.2 where the Interceptor monitor
was in use (rows 5 - 12) and **response time** of **book_flight** was monitored (rows 5 and 7-12). For each of these, Line 8 checks that monitoring setting knowledge exists for each monitoring setting in use (which is true for all rows 5 - 12). Line 9 then checks that overhead knowledge exists for each monitor in use (except the monitor of interest). Since we have no overhead knowledge of the **Eavesdropper** monitor in this case, we are restricted to rows 5 and 7. For row 5, the set of monitoring setting impacts (4%) is summed at Line 12, and the set of overhead impacts from other monitors (0%) is summed at line 15. The delta between the sum of these impacts and the recorded impact at row 5 in Table 7.2 (4% - 4% = 0%) is added to the set of **deltas** (Line 17). Similarly for row 7, the total recorded impact (8%) is taken from the set of monitoring setting impacts (8%) (8% - 8% = 0%) and added to the **deltas**. The average delta is then 0%, which is the overhead impact of the **Interceptor** monitor on the **response time** of **book_flight**.

### 7.2.2.4 Summary

After execution, the **ms_kn** array contains knowledge about the impacts of individual monitoring settings’ monitoring levels, on each QoS, and the **oh_kn** array contains knowledge about the overhead impacts that each monitor has on each QoS. Note that it is possible that not all impact values can be calculated if there is not a wide enough range of QoS records to analyse. Specifically, we require at least one mirrored monitoring configuration for each monitoring setting in order to determine the impact of that setting. Impact levels are set to 0 if they could not be determined due to insufficient information. In cases with insufficient monitoring records, we may use old knowledge about monitoring impacts, or modify monitoring configurations that are required to determine monitoring impacts, if it is valuable and possible to do so. That is, we may manually control monitors for impact analysis if there is a sufficiently small business cost in disabling and enabling a monitor during run-time. This would allow for the impacts of any monitor to be established, providing requirements for monitoring allow for monitoring configurations to be controlled for the purpose of learning monitoring impacts.

Experiments measuring the accuracy achieved by this impact analysis algorithm for a case study system are described in Section 7.3.
7.3 Experiments and Results

We have designed a series of experiments to measure the effectiveness of our technique under various noise levels and sample sizes. The experiments are based on an extension to the fictional example system described in Section 7.2, as depicted in Figure 7.3. This extension is provided to ensure there are enough parameters in the problem set to make impact analysis realistic. This system consists of one service, three monitors, and four quality types. Since our technique measures the impacts on each service-quality type pair separately, adding more than one impacted service-quality type pair to the scenario will not increase the complexity of the problem. Therefore, we have focussed on the number of monitoring settings (in this case 10, one for each quality type that each monitor observes) that may cause impacts.

Based on this scenario, we generated QoS monitoring record sets. Each record set consists of between 2,000 and 20,000 pairs of individual monitoring configuration and QoS results, in the same format as Table 7.1. For each result in a record set, the monitoring configuration was randomly generated (monitoring settings were randomly enabled or disabled), and the resulting QoS impacts of using that monitoring configuration were calculated, based on the given monitoring costs (used to generate the sets of monitoring records). Noise was then added to these impact levels to simulate external noise. For our QoS records, one result represents one monitored event (e.g. a service execution) under a particular monitoring configuration. The noise level is used to scale the noise for each record: $\text{Final\_QoS} = \text{QoS} + \text{normal\_rand}(\text{QoS}, (\text{noiselevel} \times \text{QoS}))$. That is, for each QoS impact in each record, a random, normally distributed value was generated with a mean of that QoS level (e.g. 100) and standard deviation of that QoS level multiplied by the given noise level. Then, the resulting noise was added to the original QoS impact.

We generated a set of QoS records for each sample size between 2,000 and 20,000, and noise levels between 0 and 0.1 with a 0.01 gradation. This provided sample sets with a range of noise from 0, representing ideal conditions with no noise, to 0.1, representing very noisy data with approximately twice as much noise as signal. Here, a noise level of 0.1 means noise is added to the ideal QoS (which is regarded as 100% for our tests) with a standard deviation of $(0.1 \times 100) = 10$. This noise is added to the QoS impact levels for each QoS record.
Table 7.3: Experimental Scenario Impacts

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Quality Measured</th>
<th>Monitoring Level</th>
<th>Quality Impacted</th>
<th>Impact Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probe</td>
<td>response time</td>
<td>1</td>
<td>response time</td>
<td>1</td>
</tr>
<tr>
<td>Probe</td>
<td>Correctness</td>
<td>1</td>
<td>response time</td>
<td>1</td>
</tr>
<tr>
<td>Probe</td>
<td>Privacy</td>
<td>1</td>
<td>response time</td>
<td>1</td>
</tr>
<tr>
<td>Probe</td>
<td>Security</td>
<td>1</td>
<td>response time</td>
<td>1</td>
</tr>
<tr>
<td>Interceptor</td>
<td>response time</td>
<td>1</td>
<td>response time</td>
<td>2</td>
</tr>
<tr>
<td>Eavesdropper</td>
<td>response time</td>
<td>1</td>
<td>response time</td>
<td>0.5</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead Impacts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Probe</td>
<td></td>
<td></td>
<td>response time</td>
<td>2.000</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td>response time</td>
<td>10.000</td>
</tr>
</tbody>
</table>
The implementation details of the Impact Analyser are described in Section 8.2. Using this implementation, we have performed evaluation of the analysis system in order to determine how accurately impact analysis can be performed under various noise and sample size conditions. In order for impact analysis to be practical, accurate results should be obtainable from noisy, moderately sized samples. Since different real-life scenarios would require different levels of accuracy with different sample sizes, we do not give a target level of accuracy for analysis that will always be sufficient. However, we assume that if a QoS in a system varies more from external noise than it does from monitoring impact, determining that impact to a high degree of accuracy is not valuable. For example, if the response time of a service varies between 10 and 20 seconds, determining that a monitor impacts that service by 1 second is not likely to be useful to the service provider. As a point of comparison, the impact levels in our test system are as low as 0.5%, which has a similar magnitude as a noise level of 0.01.

We measure the average error of our technique using two metrics. The first metric, Prediction Error, measures how effective impact analysis is by using learned impacts (from performing analysis) to predict the net impact of different monitoring configurations. The second metric, Raw Error, measures how effective impact analysis is by comparing learned impacts against the actual impacts used to generate the monitoring records analysed.
7.3.1 Prediction Error

For the first error metric, we performed 2-fold cross-validation by analysing half of the records (between 1,000 and 10,000) for each record set, with the remaining records used to measure the error in the analysis. The accuracy of the analysed monitoring impacts was measured after analysis, by scoring the analysed monitoring impacts against the validation data. The error levels reported are the average error across all validation records in fold (the set of un-used raw monitoring records), where the error for an individual record \( r \in \text{fold} \) is the percentage difference between the original QoS impact in the record \( \text{impact}(r) \) and the QoS impact calculated using the record’s monitoring configuration and analysed impact levels. That is, for the set of monitoring records \( \text{fold} \) and sets of analysed monitoring setting impacts \( \text{MSI}_l \), and analysed monitoring overhead impacts \( \text{MO}_l \),

\[
\text{prediction_error} = \sum_{r \in \text{fold}} \text{abs}(\text{impact}(r) - \text{predicted_impact}(r));
\]

\[
\text{predicted_impact}(r) = \sum_{m \in r} \text{MQI}_l (m) + \sum_{m \in r} \text{MO}_l (m));
\]

Figure 7.4 gives the error for our analysis technique for a set of experiments on our case study system. The figure shows an average error measure for our case study system, with noise levels between 0 and 0.1, and sample sizes between 1,000 and 10,000. For each sample size and noise level, we generated 10 sets of test data. The reported error is the average error over these noise levels, sample sizes, both quality types, and 10 test instances. The error levels reported give the average of the difference (as a percentage) between each record’s reported QoS and the QoS calculated using the analysed impact levels, as defined above. In this case an error level of 5% means that the average difference between the impact levels from the QoS records validation (fold) data and the impact levels derived on those same QoS records using the analysed impact levels was 5%. Figure 7.4 shows that the average error using 1,000 samples rises approximately linearly from 0% to 20% as the noise level increases from 0 to 0.1. However, as sample sizes increase, the average error stays below 5%. For sample sizes of 4,000 and 10,000 QoS records, the average errors for noise level 0.1 were 2.8% and 0.8%, respectively. At a noise level of 0.01, we require a sample size of 4,000 to ensure an error level below 0.5%.
Figure 7.4: Impact Analysis Prediction Error
7.3.2 Raw Error

For the second error metric, we used the same data and configuration as that described in Section 7.3.1. We analysed half of the records (between 1,000 and 10,000) for each record set, and calculated the difference between the analysed impacts and the actual impacts for the system. That is, for the sets of all actual monitoring setting impacts MQI and monitoring overhead impacts MO (those in Table 7.3); and sets of analysed monitoring impacts MQI_l and analysed monitoring overhead impacts MO_l,

\[
\text{raw impacts error} = \sum_{ms \in MS} \text{abs}(MQI(ms)) - (MQI_l(ms)) + \sum_{m \in M} \text{abs}((MO(m)) - (MO_l(m))
\]

Figure 7.5 shows the raw error for our set of tests. The error levels reported give the average of the difference (as a percentage) between each reported impact level and the actual impact level used to generate the sample set. For example, if a set of impact levels for a test set was \(\{S_1 M_1 Q_1 \rightarrow (S_1 Q_1, 10\%); M_1.oh \rightarrow (S_1 Q_1, 5\%)}\), and the set of analysed impacts was \(\{S_1 M_1 Q_1 \rightarrow (S_1 Q_1, 8\%); M_1.oh \rightarrow (S_1 Q_1, 7\%)}\), then the average percentage error would be 30\%. This is because the analysed impact for \(S_1 M_1 Q_1\) was 8\% versus the real value of 10\% (giving 20\% error), and the analysed impact for \(M_1.oh\) was 7\% versus the real value of 5\% (giving 40\% error). To ensure consistency with the errors reported in Figure 7.4 the set of QoS records was still folded, with only half of each record set used for analysis. Figure 7.5 shows that the average error using 1,000 samples rises approximately linearly from 0\% to 60\%, as the noise level increases from 0 to 0.1. Using the largest sample size (10,000), the average error rose from 0\% to 15\%, as the noise level increased from 0 to 0.1. Note that while these percentages seem to indicate higher error levels than those in Figure 7.4, these errors are measured against smaller values (e.g. impacts of 1\%), where a variation to just 1.1\% gives a 10\% error. At a noise level of 0.01, we require a sample size of 1,000 to ensure an error level below 0.5\%.

7.3.3 Discussion

Overall our results demonstrate that it is possible to determine the impacts of individual monitors from the integrated impacts of a set of monitors in a Web Service system, using only existing monitoring logs from that Web Service system. Having a large enough set of monitoring records with a range of monitoring configurations allows for monitoring
Figure 7.5: Impact Analysis Raw Error

Average Learned Impacts Error (%)

Noise Level

1000
2000
4000
6000
8000
10000
impacts to be determined with a high degree of accuracy (an error level lower than the system’s natural noise level). It should be noted that if there are not enough monitoring configurations used, then analysis may not always yield results for all impacted Qualities of Service. As discussed, this may be overcome by manually configuring systems for the purpose of measuring monitoring impacts if the costs of modifying the monitoring configuration for the sole purpose of impact analysis are not too high. Monitoring impact knowledge may then be used to balance monitoring coverage and monitoring impacts via informed reconfiguration of a Web Service monitoring system [24], or reconfiguration of host systems by performing actions such as adding more hardware to support monitors that are causing Service Level Agreements (SLAs) to be breached.

7.4 Related Work

Zhang et al. describe a technique for using regression-based analysis to estimate the CPU demand of customer transactions on multi-tier systems [67]. This is used to ensure QoS levels are met whilst performing dynamic resource provisioning. Whilst analysing records to determine individual impact levels in an aggregated net impact, the authors are measuring CPU demand in a specific system configuration rather than generic QoS impacts. Rolia and Vetland describe a method for using multi-linear regression analysis to estimate server resource demand from parallel services, using monitoring logs [56]. This work uses logs of service loads and response times to estimate server resource demands.

Katchabaw et al. describe a method for evaluating the cost of management functions in distributed systems [26]. This work uses a set of generic tools and scripts to enable the automated profiling of a distributed management system. The tools may be used to deploy, re-deploy, and configure monitors automatically, whilst performing system tests on both resource usage (e.g. CPU and network bandwidth) and QoS (e.g. response time and error rate). There is similar work in testing the performance impact monitoring in J2EE applications [49]. Whilst these methods allow for the impacts of monitoring on a system to be determined, they are designed to do so at design-time or deployment-time, by actively modifying and testing a set of services. In contrast, our technique allows for the impact of monitors to be determined without actively enabling and disabling those monitors and performing experiments on the system under evaluation.
Whilst there has been work that analyses monitoring logs to measure server resource demand of services or transactions, we have discovered no work that uses monitoring logs to determine the impact of monitors on a system. Furthermore, we have discovered no work that models monitors as sets of capabilities and determines their impacts according to their actions.

7.5 Summary

We have described a technique for using run-time logs from a suite of Web Service monitors to determine the impact of individual monitors on any delivered QoS, according to the actions that each monitor is performing. This technique analyses logs that provide detail of a monitoring configuration for the system along with the QoSs that were delivered under that configuration. By comparing the delivered QoS between different monitoring configurations, we can determine the impact levels of each monitor. These impact levels include overhead impact for a monitor and a set of monitoring setting impacts that describe the impacts of each individual monitoring action. We described a case study system and generated a set of monitoring logs for that system, which varied in size and noise level. These logs were then analysed to determine the impacts of monitors and monitoring settings in the system. Our approach was able to determine the impacts of monitors to a high degree of accuracy, where the error in our estimation was lower than the noise level in the monitoring logs.
Chapter 8

Case Study

Recall the iTravel scenario from Chapter 2, in which iTravel was a provider of web services to travel agents for searching and booking flights. These services are offered with Service Level Agreements (SLAs) providing guarantees on qualities such as response time, and are bound by corporate governance and legal requirements that ensure services are monitored for aspects such as privacy and security. Each of these monitors has certain impacts on the iTravel services, and iTravel wishes to reduce these impacts whilst maintaining acceptable monitoring coverage.

The iTravel scenario described in Chapter 2 has been extended for use as a case study system. The new iTravel case study system is described in Section 8.1. This new system is used to demonstrate the application of our optimisation and impact analysis techniques on a realistic system, as well as measure the effectiveness of our techniques on that system. After describing the iTravel case study system and its attributes, we present results of experiments on the design-time optimisation, impact analysis, and run-time optimisation of the iTravel case study system.

Section 8.1 describes the extended iTravel case study. Section 8.2 describes the implementation of the system, including the optimisation framework. Section 8.3 describes the results of design-time optimisation of this system. Section 8.4 describes the results of once-off and continuous run-time optimisation of the system. Section 8.5 describes the results of performing impact analysis on the system.
8.1 System Description

The following sections describe the iTravel case study’s system structure, requirements, monitors and monitoring capabilities, baseline QoS levels, and the impacts of monitors in the system.

8.1.1 System Structure

Figure 8.1 depicts the iTravel case study system. The system is divided into flight, hotel, accounting and administration subsystems. The flight and hotel subsystems each consist of an ‘info’ and a ‘book’ service, the accounting subsection includes a ‘billing’ service, which uses either of a charge_credit_card or debit_account service for billing, and the administration subsection includes an admin and a marketing service.

iTravel has a set of requirements that are not considered met unless their respective Qualities of Service (Qualities of Service) are monitored. As such, the flight and hotel subsystems each include two probe-based monitors, two interceptor-based monitors, and one eavesdropper-based monitor. The accounting subsection includes a probe-based monitor and an interceptor-based monitor. The administration subsection includes two interceptor-based monitors and two probe-based monitors.

The accounting and administration subsystems are hosted on a shared server, whilst the flight and hotel subsystems are each hosted on a dedicated server. As shown in Figure 8.1, each subsystem’s monitors are hosted on that subsystem’s server. For example, the flight_info Probe, book_flight Probe, flight_info Interceptor, book_flight Interceptor and Generic Flight Eavesdropper are all hosted on the same server as the flight_info and book_flight services. Table 8.1 shows the hardware used for each system in our experiments.

<table>
<thead>
<tr>
<th>Server</th>
<th>CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight Subsystem</td>
<td>Intel E8500 @ 3.16GHz</td>
<td>4GB</td>
</tr>
<tr>
<td>Hotel Subsystem</td>
<td>Intel T7700 @ 2.4GHz</td>
<td>3GB</td>
</tr>
<tr>
<td>Accounting and Administration Subsystems</td>
<td>T5600 @ 1.8GHz</td>
<td>1GB</td>
</tr>
</tbody>
</table>

Table 8.1: Server Specifications
Figure 8.1: iTravel Case Study System
8.1.2 Requirements

iTravel is bound by SLA, business, and legal requirements, and has a set of QoS goals. The SLA requirements are from a single SLA that all service consumers have with iTravel, and the business and legal requirements come from iTravel’s business and legal obligations. The QoS goals are self-imposed and reflect the value of delivering particular QoS levels to consumers. These requirements are described in the following sections.

iTravel’s QoS requirements are summarised in Table 8.2. Each of the rows in Table 8.2 defines one requirement for iTravel. Each requirement consists of an ID (e.g. B1), along with a service (e.g. flight info) and quality type (e.g. Security) and required quality level (e.g. 100%). Each requirement also includes the penalties to iTravel for non-compliance.

Some of the requirements are fixed, such as business requirement B1, which proscribes a fixed penalty for breaches. Others, such as the SLA-sourced requirements and Goal-based requirements link to penalties based on the number of consumers and their account fees. Some requirements are also graded, such as the goals G5 to G8, which represent the decreasing value to iTravel as response times increase. In this case, the value of providing a response time between four and eight seconds is worth $0.25 per consumer, whilst the value of providing a response time below two seconds is worth $1 per consumer.

Penalties in Table 8.2 are each listed as a penalty directly derived from a requirement (e.g. the penalty for requirement G1 is “$0.50 PC”) as well as a total penalty ($2,500 for this example), which is the total amount based on the number of consumers and their account fees. Total penalties have been translated into utility levels as described in Chapter 5.4.4.
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Service</th>
<th>Quality</th>
<th>Will Be</th>
<th>Penalty</th>
<th>Total Penalty</th>
<th>Utility</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>flight_info</td>
<td>Security</td>
<td>100%</td>
<td>$30,000.00</td>
<td>$30,000.00</td>
<td>0.0273</td>
</tr>
<tr>
<td>B2</td>
<td>hotel_info</td>
<td>Security</td>
<td>100%</td>
<td>$30,000.00</td>
<td>$30,000.00</td>
<td>0.0273</td>
</tr>
<tr>
<td>B3</td>
<td>book_flight</td>
<td>Security</td>
<td>100%</td>
<td>$100,000.00</td>
<td>$100,000.00</td>
<td>0.0909</td>
</tr>
<tr>
<td>B4</td>
<td>book_hotel</td>
<td>Security</td>
<td>100%</td>
<td>$100,000.00</td>
<td>$100,000.00</td>
<td>0.0909</td>
</tr>
<tr>
<td>B5</td>
<td>admin</td>
<td>Security</td>
<td>100%</td>
<td>$200,000.00</td>
<td>$200,000.00</td>
<td>0.1818</td>
</tr>
<tr>
<td>B6</td>
<td>marketing</td>
<td>Security</td>
<td>100%</td>
<td>$200,000.00</td>
<td>$200,000.00</td>
<td>0.1818</td>
</tr>
<tr>
<td>B7</td>
<td>billing</td>
<td>Security</td>
<td>100%</td>
<td>$200,000.00</td>
<td>$200,000.00</td>
<td>0.1818</td>
</tr>
<tr>
<td>B8</td>
<td>flight_info</td>
<td>DLP</td>
<td>100%</td>
<td>$50,000.00</td>
<td>$50,000.00</td>
<td>0.0455</td>
</tr>
<tr>
<td>B9</td>
<td>hotel_info</td>
<td>DLP</td>
<td>100%</td>
<td>$50,000.00</td>
<td>$50,000.00</td>
<td>0.0455</td>
</tr>
<tr>
<td>B10</td>
<td>book_flight</td>
<td>DLP</td>
<td>100%</td>
<td>$50,000.00</td>
<td>$50,000.00</td>
<td>0.0455</td>
</tr>
<tr>
<td>B11</td>
<td>book_hotel</td>
<td>DLP</td>
<td>100%</td>
<td>$50,000.00</td>
<td>$50,000.00</td>
<td>0.0455</td>
</tr>
<tr>
<td>B12</td>
<td>marketing</td>
<td>DLP</td>
<td>100%</td>
<td>$50,000.00</td>
<td>$50,000.00</td>
<td>0.0455</td>
</tr>
<tr>
<td>G1</td>
<td>flight_info</td>
<td>Reliability</td>
<td>≥95%</td>
<td>$0.50 PC</td>
<td>$2,500.00</td>
<td>0.0023</td>
</tr>
<tr>
<td>G2</td>
<td>hotel_info</td>
<td>Reliability</td>
<td>≥95%</td>
<td>$0.50 PC</td>
<td>$2,500.00</td>
<td>0.0023</td>
</tr>
<tr>
<td>G3</td>
<td>book_flight</td>
<td>Reliability</td>
<td>≥98%</td>
<td>$1.00 PC</td>
<td>$5,000.00</td>
<td>0.0045</td>
</tr>
<tr>
<td>G4</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>≥98%</td>
<td>$1.00 PC</td>
<td>$5,000.00</td>
<td>0.0045</td>
</tr>
<tr>
<td>G5</td>
<td>book_flight</td>
<td>Response Time</td>
<td>≤2s</td>
<td>$1.00 PC</td>
<td>$5,000.00</td>
<td>0.0014</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Service</th>
<th>Quality</th>
<th>Will Be</th>
<th>Penalty</th>
<th>Total Penalty</th>
<th>Utility</th>
</tr>
</thead>
<tbody>
<tr>
<td>G6</td>
<td>book_flight</td>
<td>Response Time</td>
<td>≤2.5s</td>
<td>$0.75 PC</td>
<td>$3,750.00</td>
<td>0.0014</td>
</tr>
<tr>
<td>G7</td>
<td>book_flight</td>
<td>Response Time</td>
<td>≤4s</td>
<td>$0.50 PC</td>
<td>$2,500.00</td>
<td>0.0014</td>
</tr>
<tr>
<td>G8</td>
<td>book_flight</td>
<td>Response Time</td>
<td>≤8s</td>
<td>$0.25 PC</td>
<td>$1,250.00</td>
<td>0.0011</td>
</tr>
<tr>
<td>G9</td>
<td>flight_info</td>
<td>Response Time</td>
<td>≤2s</td>
<td>$1.00 PC</td>
<td>$5,000.00</td>
<td>0.0011</td>
</tr>
<tr>
<td>G10</td>
<td>flight_info</td>
<td>Response Time</td>
<td>≤2.5s</td>
<td>$0.75 PC</td>
<td>$3,750.00</td>
<td>0.0011</td>
</tr>
<tr>
<td>G11</td>
<td>flight_info</td>
<td>Response Time</td>
<td>≤4s</td>
<td>$0.50 PC</td>
<td>$2,500.00</td>
<td>0.0011</td>
</tr>
<tr>
<td>G12</td>
<td>flight_info</td>
<td>Response Time</td>
<td>≤8s</td>
<td>$0.25 PC</td>
<td>$1,250.00</td>
<td>0.0011</td>
</tr>
<tr>
<td>G13</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>≤2s</td>
<td>$1.00 PC</td>
<td>$5,000.00</td>
<td>0.0014</td>
</tr>
<tr>
<td>G14</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>≤2.5s</td>
<td>$0.75 PC</td>
<td>$3,750.00</td>
<td>0.0014</td>
</tr>
<tr>
<td>G15</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>≤4s</td>
<td>$0.50 PC</td>
<td>$2,500.00</td>
<td>0.0014</td>
</tr>
<tr>
<td>G16</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>≤8s</td>
<td>$0.25 PC</td>
<td>$1,250.00</td>
<td>0.0014</td>
</tr>
<tr>
<td>G17</td>
<td>hotel_info</td>
<td>Response Time</td>
<td>≤2s</td>
<td>$1.00 PC</td>
<td>$5,000.00</td>
<td>0.0011</td>
</tr>
<tr>
<td>G18</td>
<td>hotel_info</td>
<td>Response Time</td>
<td>≤2.5s</td>
<td>$0.75 PC</td>
<td>$3,750.00</td>
<td>0.0011</td>
</tr>
<tr>
<td>G19</td>
<td>hotel_info</td>
<td>Response Time</td>
<td>≤4s</td>
<td>$0.50 PC</td>
<td>$2,500.00</td>
<td>0.0011</td>
</tr>
<tr>
<td>G20</td>
<td>hotel_info</td>
<td>Response Time</td>
<td>≤8s</td>
<td>$0.25 PC</td>
<td>$1,250.00</td>
<td>0.0011</td>
</tr>
<tr>
<td>L1</td>
<td>book_hotel</td>
<td>Privacy Act 1</td>
<td>100%</td>
<td>$100,000.00</td>
<td>$100,000.00</td>
<td>0.0909</td>
</tr>
<tr>
<td>L2</td>
<td>marketing</td>
<td>Privacy Act 2</td>
<td>100%</td>
<td>$200,000.00</td>
<td>$200,000.00</td>
<td>0.1818</td>
</tr>
<tr>
<td>L3</td>
<td>marketing</td>
<td>Spam Act</td>
<td>100%</td>
<td>$1,100,000.00</td>
<td>$1,100,000.00</td>
<td>1.0000</td>
</tr>
<tr>
<td>L4</td>
<td>flight_info</td>
<td>Privacy Act 3</td>
<td>100%</td>
<td>$50,000.00</td>
<td>$50,000.00</td>
<td>0.0455</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Service</th>
<th>Quality</th>
<th>Will Be</th>
<th>Penalty</th>
<th>Total Penalty</th>
<th>Utility</th>
</tr>
</thead>
<tbody>
<tr>
<td>L5</td>
<td>hotel_info</td>
<td>Privacy Act 3</td>
<td>100%</td>
<td>$50,000.00</td>
<td>$50,000.00</td>
<td>0.0455</td>
</tr>
<tr>
<td>L6</td>
<td>book_flight</td>
<td>Privacy Act 3</td>
<td>100%</td>
<td>$50,000.00</td>
<td>$50,000.00</td>
<td>0.0455</td>
</tr>
<tr>
<td>L7</td>
<td>book_hotel</td>
<td>Privacy Act 3</td>
<td>100%</td>
<td>$50,000.00</td>
<td>$50,000.00</td>
<td>0.0455</td>
</tr>
<tr>
<td>L8</td>
<td>marketing</td>
<td>Privacy Act 3</td>
<td>100%</td>
<td>$50,000.00</td>
<td>$50,000.00</td>
<td>0.0455</td>
</tr>
<tr>
<td>SLA1</td>
<td>flight_info</td>
<td>Response Time</td>
<td>≤10s</td>
<td>33% AF</td>
<td>$3,300.00</td>
<td>0.0030</td>
</tr>
<tr>
<td>SLA2</td>
<td>hotel_info</td>
<td>Response Time</td>
<td>≤10s</td>
<td>33% AF</td>
<td>$3,300.00</td>
<td>0.0030</td>
</tr>
<tr>
<td>SLA3</td>
<td>book_flight</td>
<td>Response Time</td>
<td>≤10s</td>
<td>66% AF</td>
<td>$6,600.00</td>
<td>0.0060</td>
</tr>
<tr>
<td>SLA4</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>≤10s</td>
<td>66% AF</td>
<td>$6,600.00</td>
<td>0.0060</td>
</tr>
<tr>
<td>SLA5</td>
<td>flight_info</td>
<td>Reliability</td>
<td>≥98%</td>
<td>100% AF</td>
<td>$10,000.00</td>
<td>0.0091</td>
</tr>
<tr>
<td>SLA6</td>
<td>book_flight</td>
<td>Reliability</td>
<td>≥98%</td>
<td>100% AF</td>
<td>$10,000.00</td>
<td>0.0091</td>
</tr>
<tr>
<td>SLA7</td>
<td>hotel_info</td>
<td>Reliability</td>
<td>≥98%</td>
<td>100% AF</td>
<td>$10,000.00</td>
<td>0.0091</td>
</tr>
<tr>
<td>SLA8</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>≥98%</td>
<td>100% AF</td>
<td>$10,000.00</td>
<td>0.0091</td>
</tr>
</tbody>
</table>

Total Penalty: $2,884,800.00
Utility: 2.5974
8.1.2.1 SLA Requirements

iTravel provides a single SLA to all of its fee-paying travel agent consumers. This SLA provides guarantees on **response time** and **reliability**.

**Response time** monitoring measures the round-trip time for a service request, on the service provider’s side. The SLA provides guarantees that **response time** will be below 10 seconds for all hotel and flight services. If this is not the case for the **flight_info** or **hotel_info** services, then consumers (of which there are 1,000) will receive a 33% refund of their monthly account fee (which is $100), and if this is not the case for the **book_flight** or **book_hotel** services, then consumers will receive a 66% refund of their account fee.

**Reliability** monitoring checks that all consumer requests receive a response, and that the response is properly formulated. The SLA provides guarantees that **reliability** will be at least 98% for the **flight_info**, **hotel_info**, **book_flight** and **book_hotel** services. If this guarantee is not met for a service, then consumers are refunded 100% of their monthly account fee. It should be noted that if more than one requirement is not met, then service consumers may receive penalties above 100% of their account fees. For example, if the reliability requirements of **book_flight** and **flight_info** are both breached, consumers will receive a refund of 200% of their account fees. This agreement to refund more than a consumer’s monthly account fee is not unique to iTravel, as demonstrated by SLAs in our SLA Survey in Appendix A (particularly, the NetSuite SLA).

Table 8.2 lists all of the requirements on the iTravel system.

8.1.2.2 Public Consumer QoS Goals

In addition to travel agent consumers, iTravel offers their service to the public, which means that on top of their travel agency consumers, they have many thousands of public consumers. Since these public consumers receive the service without an account fee, they do not receive an SLA with monetary penalties. However, to retain customers the iTravel project team have imposed **service level goals** for the publicly available services as retaining such public customers through satisfactory QoS is regarded as delivering particular business value (through advertising revenue). These requirements are on the **reliability** and **response time** of iTravel’s services.

The **reliability** goals (G1 to G4 in Table 8.2) are fixed, with iTravel considering the
flight_info and hotel_info services as requiring 95% or higher reliability with a value of $0.5 per consumer, and the book_flight and book_hotel services as requiring 98% or higher reliability with a value of $1 per consumer. Although the penalties for the goals are not actually paid out by iTravel, they represent the value to iTravel of achieving those service levels. The value is calculated per consumer, and the number of consumers is estimated as the average number of monthly public users, which for iTravel is currently 5,000.

The response time goals (G5 to G20 in Table 8.2) are graded from 2 seconds to 8 seconds, and represent the decreasing value to iTravel as response times increase. For example, requirements G5 to G8 describe the value to iTravel for response times between 2 and 8 seconds. In this case, the value for response times less than or equal to 2 seconds is $1.00 per consumer, the value for less than or equal to 2.5 seconds is $0.75 per consumer, the value for less or equal to 4 seconds is $0.50 per consumer, and the value for less than or equal to 8 seconds is $0.25 per consumer (there is no value to iTravel for response times over 8 seconds).

8.1.2.3 Business Requirements

iTravel has a set of business-level requirements that must be met. These requirements concern security and Data Leak Protection (DLP) of all services offered, and reflect the value iTravel places on customer satisfaction for these attributes.

Security monitoring (requirements B1 to B7 in Table 8.2) supports the assessment of the compliance with security regulations by continuously producing well-defined and traceable evidence (hereafter simplified to ‘Security’) by checking messages for known weaknesses such as Structured Query Language (SQL) injection and verifying that actions such as authentication have been performed. The value to iTravel for monitoring security is: $30,000 for each of the flight_info and hotel_info services; $100,000 for each of the book_flight and book_hotel services; $200,000 for each of the admin and marketing services; and $200,000 for the billing service. iTravel considers either probe-based security monitoring or interceptor-based security monitoring methods as appropriate means to validate security. It should be noted that security monitoring provides indicators or metrics only, i.e. security monitoring is not a guarantee that security has been met.
However, iTravel has deemed its security monitors as sufficient indicators for security in the iTravel system.

DLP monitoring (requirements B8 and B9 in Table 8.2) is used to ensure that all outgoing messages contain no un-encrypted sensitive information, and that no sensitive information is being leaked or sent to inappropriate parties. This is achieved by performing actions such as verifying and cleansing outgoing personal information. The value to iTravel for performing DLP monitoring is $50,000 for each publicly facing service. iTravel considers interceptor-based monitoring as the only appropriate method to validate DLP, since firewalling actions must be performed.

The penalties for these requirements are fixed rather than per-consumer. This means that whilst the individual penalties seem high, the penalties from SLAs may have a higher total cost if there are a large enough number of consumers.

8.1.2.4 Legal Requirements

iTravel operates in Australia, and is bound by Australian regulations on privacy (Requirements L1, L2, and L4 to L8 in Table 8.2) and spam (Requirement L3 in Table 8.2), for all services that it offers.

The Privacy Act 1988 [1] sets out a list of privacy obligations for Australian businesses, along with a set of ‘Privacy Principles’ that are guidelines for businesses to meet these obligations. The relevant obligations to iTravel are:

(1) Only collect personal information that is necessary for your functions or activities

(2) Only use or disclose personal information for the primary purpose of collection, unless by consumer consent

(3) Take reasonable steps to protect the personal information you hold from misuse and loss and from unauthorised access, modification or disclosure

Whilst failure to meet these obligations does not impose direct fines or penalties, civil action may apply\(^1\). Therefore, iTravel has analysed legal precedent and placed value on monitoring to ensure that these privacy obligations are met. The following monitoring activities are required in order to ensure that the privacy act is met:

(1) The book_hotel service must be monitored to ensure that personal information not necessary for booking a hotel, including visa numbers or passport numbers, are not transmitted to or from the book_hotel service, as this would breach the obligation. Legal analysis suggests that the value to iTravel for monitoring to ensure this requirement is met is $100,000.

(2) The marketing service must be monitored to ensure that any incoming or outgoing personal information is linked to a consumer account for a consumer who has agreed to their information being used for marketing purposes. This monitoring requirement is used to ensure that obligation 2 is not breached. Legal analysis suggests that the value to iTravel for monitoring to ensure this requirement is met is $200,000.

(3) All publicly facing services must be monitored to ensure that outgoing personal information is not sent un-encrypted. This monitoring requirement is used to ensure that obligation 3 is not breached. Legal analysis suggests that the value to iTravel for monitoring to ensure this requirement is met is $50,000 per publicly facing service.

The Spam Act 2003 [2] requires Australian businesses to ensure that they do not send unsolicited, commercial marketing messages via e-mail. The penalty for breaching the Spam Act is up to AUD$1.1 million. Therefore, iTravel has determined that monitoring outgoing marketing messages to ensure that the Spam Act is not breached has a value of AUD$1.1 million. This monitoring requires the marketing service to be monitored in order to ensure that for every outgoing message, the recipient is listed as having consented to receive promotional messages.

### 8.1.3 Monitoring Capabilities

The monitors and their monitoring capabilities for the iTravel system are summarised in Table 8.3, showing which Monitors are capable of monitoring which Quality Types of which Services. In all cases, monitoring capabilities have two potential monitoring levels (completely disabled and completely enabled).

The billing, flight and hotel probe-based monitors (billing Probe, flight_info Probe, book_flight Probe, hotel_info Probe, and book_hotel Probe) are capable of meeting requirements for monitoring response time, reliability and security of their target Web...
Services by invoking the services and pretending to be service consumers. The administration probe monitors (admin Probe and marketing Probe) are capable of meeting requirements for monitoring security. Each probe is hosted on the same server as its target Web Services.

The flight and hotel interceptor-based monitors (flight_info Interceptor, book_flight Interceptor, hotel_info Interceptor, and book_hotel Interceptor) are all capable of monitoring security, response time, reliability and DLP, whilst the book_hotel Interceptor can also monitor for privacy act obligation 1. The admin Interceptor can monitor security and DLP. The Accounting Interceptor can monitor security, response time and DLP. The marketing Interceptor can monitor security, DLP, privacy act obligation 2, and the spam act obligation. The interceptor-based monitors do this by intercepting the service requests of consumers, rather than generating their own requests as probes do. The interceptors have the capability to forward messages after analysing them, and therefore interceptors can filter or modify communications and perform functions such as firewalling. The interceptor monitors are hosted on the same servers as their target services. It should be noted that monitoring for DLP to meet business-level requirements subsumes the monitoring requirement to meet Privacy Act Obligation 3. That is, monitoring for DLP for a service covers both of these requirements.

The eavesdropper monitors (the Generic Hotel Eavesdropper and Generic Flight Eavesdropper) act similarly to the interceptors; however, they cannot modify or stop a message from being transmitted, as they are only passive listeners. The Generic Eavesdroppers are hosted on their own, dedicated server.

Table 8.3: iTravel Monitoring Capabilities

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Service</th>
<th>Quality Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting Interceptor</td>
<td>billing</td>
<td>DLP</td>
</tr>
<tr>
<td>Accounting Interceptor</td>
<td>billing</td>
<td>Response Time</td>
</tr>
<tr>
<td>Accounting Interceptor</td>
<td>billing</td>
<td>Security</td>
</tr>
<tr>
<td>admin Interceptor</td>
<td>admin</td>
<td>DLP</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Monitor</th>
<th>Service</th>
<th>Quality Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin Interceptor</td>
<td>admin</td>
<td>Security</td>
</tr>
<tr>
<td>admin Probe</td>
<td>admin</td>
<td>Security</td>
</tr>
<tr>
<td>billing Probe</td>
<td>billing</td>
<td>Reliability</td>
</tr>
<tr>
<td>billing Probe</td>
<td>billing</td>
<td>Response Time</td>
</tr>
<tr>
<td>billing Probe</td>
<td>billing</td>
<td>Security</td>
</tr>
<tr>
<td>book_flight Probe</td>
<td>book_flight</td>
<td>Reliability</td>
</tr>
<tr>
<td>book_flight Probe</td>
<td>book_flight</td>
<td>Response Time</td>
</tr>
<tr>
<td>book_flight Probe</td>
<td>book_flight</td>
<td>Security</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>DLP</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>Reliability</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>Response Time</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>Security</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>Reliability</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>Response Time</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>Security</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>DLP</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>Privacy Act Obligation 1</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>Reliability</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>Response Time</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>Security</td>
</tr>
<tr>
<td>flight_info Probe</td>
<td>flight_info</td>
<td>Reliability</td>
</tr>
<tr>
<td>flight_info Probe</td>
<td>flight_info</td>
<td>Response Time</td>
</tr>
<tr>
<td>flight_info Probe</td>
<td>flight_info</td>
<td>Security</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>flight_info</td>
<td>DLP</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>flight_info</td>
<td>Reliability</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>flight_info</td>
<td>Response Time</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>flight_info</td>
<td>Security</td>
</tr>
<tr>
<td>Generic Flight Eavesdropper</td>
<td>book_flight</td>
<td>Response Time</td>
</tr>
<tr>
<td>Generic Flight Eavesdropper</td>
<td>flight_info</td>
<td>Response Time</td>
</tr>
</tbody>
</table>

Continued on Next Page...
8.1.4 Baseline QoS

The baseline response time of the flight_info, book_flight and book_hotel services is 2 seconds, and the response time of the hotel_info service is 1.5 seconds. These baseline response times are taken from benchmarks on the system under test, and involved executing all services simultaneously under a load of 10 clients per service. To achieve this, we configured a client load generator to send ten service requests to every service, and continuously maintain this load level by sending one new request to a service when that service responded to a previous request. Response times were taken as the average response time of each service, measured over 1000 service executions, and excluding the first 10 service executions from the client load generator.

The baseline reliability of all services was measured using the same process, and was measured as 100%. All other attributes are also at 100%, as long as they are being monitored.
8.1.5 Monitoring Impacts

We have measured the response time impacts that occur from monitoring in the iTravel system, and present the results in Tables 8.4 and 8.5. These impacts are an average percentage increase of response time, compared with the response time of a service without monitoring. For example, using the book_flight Interceptor will increase response time by 14% for the book_flight service, plus 25% for each property measured (security, response time, DLP, and reliability). Additionally, for each quality type measured with the book_flight Interceptor, the response time of the flight_info service is increased by 10%. All monitoring setting impacts were measured by individually enabling each monitoring capability and re-executing all services as per the baseline QoS measurements. The average differences between the resulting QoS levels were assigned to the monitoring impacts. To determine monitoring overheads, monitors were individually, manually enabled but configured to perform no monitoring, and the QoS levels were benchmarked again for each monitor.

The book_flight, flight_info, book_hotel and hotel_info Probe and Interceptor monitors all reduce the response time of their target service as reflected in Tables 8.4 and 8.5. Most of these monitors also reduce the response time of a service that is collocated with the target service (for example, the flight_info probe reduces the response time of book_flight). This is due to monitors and services sharing the same, limited set of resources.

The flight and hotel Interceptor monitors each reduce the reliability of their target service by 1% per quality type monitored.

The Generic Eavesdropper monitors have no impact on any service, and the monitors in the Administration Subsystem do not impact any of the Hotel or Flight services.

For iTravel, the set of monitoring setting impacts iMSI is shown in Table 8.4, and the set of monitoring overhead impacts iMO is shown in Table 8.5. All impacts have been derived from benchmarks on the iTravel system.
Table 8.4: iTribe Monitoring Setting Impacts

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Service Monitored</th>
<th>Quality Type Monitored</th>
<th>Service Impacted</th>
<th>Quality Type Impacted</th>
<th>Impact Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting Interceptor</td>
<td>Billing</td>
<td>Privacy</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.125</td>
</tr>
<tr>
<td>Accounting Interceptor</td>
<td>Billing</td>
<td>Privacy</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.125</td>
</tr>
<tr>
<td>Accounting Interceptor</td>
<td>Billing</td>
<td>Response Time</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.125</td>
</tr>
<tr>
<td>Accounting Interceptor</td>
<td>Billing</td>
<td>Security</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.125</td>
</tr>
<tr>
<td>Accounting Interceptor</td>
<td>Billing</td>
<td>Security</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.125</td>
</tr>
<tr>
<td>Billing Probe</td>
<td>Billing</td>
<td>Reliability</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.025</td>
</tr>
<tr>
<td>Billing Probe</td>
<td>Billing</td>
<td>Reliability</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.025</td>
</tr>
<tr>
<td>Billing Probe</td>
<td>Billing</td>
<td>Response Time</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.025</td>
</tr>
<tr>
<td>Billing Probe</td>
<td>Billing</td>
<td>Response Time</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.025</td>
</tr>
<tr>
<td>Billing Probe</td>
<td>Billing</td>
<td>Security</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.025</td>
</tr>
<tr>
<td>Billing Probe</td>
<td>Billing</td>
<td>Security</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.025</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>Privacy</td>
<td>book_flight</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>Privacy</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Monitor</th>
<th>Service Monitored</th>
<th>Quality Type Monitored</th>
<th>Service Impacted</th>
<th>Quality Type Impacted</th>
<th>Impact Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Privacy</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.100</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Reliability</td>
<td>book_flight</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Reliability</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Reliability</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.100</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Response Time</td>
<td>book_flight</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Response Time</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Response Time</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.100</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Security</td>
<td>book_flight</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Security</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Security</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.100</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Reliability</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.030</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Reliability</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.015</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Reliability</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.015</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Response Time</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.030</td>
</tr>
<tr>
<td>book_flight</td>
<td>book_flight</td>
<td>Response Time</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.015</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Monitor</th>
<th>Service Monitored</th>
<th>Quality Type Monitored</th>
<th>Service Impacted</th>
<th>Quality Type Impacted</th>
<th>Impact Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>book_flight Probe</td>
<td>book_flight</td>
<td>Response Time</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.015</td>
</tr>
<tr>
<td>book_flight Probe</td>
<td>book_flight</td>
<td>Security</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.030</td>
</tr>
<tr>
<td>book_flight Probe</td>
<td>book_flight</td>
<td>Security</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.015</td>
</tr>
<tr>
<td>book_flight Probe</td>
<td>book_flight</td>
<td>Security</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.015</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>book_hotel</td>
<td>Privacy</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>book_hotel</td>
<td>Privacy</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>book_hotel</td>
<td>Privacy</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.100</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.100</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.100</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>book_hotel</td>
<td>Security</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Monitor</th>
<th>Service Monitored</th>
<th>Quality Type Monitored</th>
<th>Service Impacted</th>
<th>Quality Type Impacted</th>
<th>Impact Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>bookHotelInterceptor</td>
<td>book_hotel</td>
<td>Security</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.100</td>
</tr>
<tr>
<td>bookHotelProbe</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.018</td>
</tr>
<tr>
<td>bookHotelProbe</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.035</td>
</tr>
<tr>
<td>bookHotelProbe</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.018</td>
</tr>
<tr>
<td>bookHotelProbe</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.018</td>
</tr>
<tr>
<td>bookHotelProbe</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.035</td>
</tr>
<tr>
<td>bookHotelProbe</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.018</td>
</tr>
<tr>
<td>bookHotelProbe</td>
<td>book_hotel</td>
<td>Security</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.018</td>
</tr>
<tr>
<td>bookHotelProbe</td>
<td>book_hotel</td>
<td>Security</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.035</td>
</tr>
<tr>
<td>bookHotelProbe</td>
<td>book_hotel</td>
<td>Security</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.018</td>
</tr>
<tr>
<td>flightInfoInterceptor</td>
<td>flight_Info</td>
<td>Privacy</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.150</td>
</tr>
<tr>
<td>flightInfoInterceptor</td>
<td>flight_Info</td>
<td>Privacy</td>
<td>flight_Info</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>flightInfoInterceptor</td>
<td>flight_Info</td>
<td>Privacy</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>flightInfoInterceptor</td>
<td>flight_Info</td>
<td>Reliability</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.150</td>
</tr>
<tr>
<td>flightInfoInterceptor</td>
<td>flight_Info</td>
<td>Reliability</td>
<td>flight_Info</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>Monitor</td>
<td>Service Monitored</td>
<td>Quality Type Monitored</td>
<td>Service Impacted</td>
<td>Quality Type Impacted</td>
<td>Impact Level</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------</td>
<td>------------------------</td>
<td>------------------</td>
<td>-----------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>flight_Info</td>
<td>Reliability</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.150</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>flight_Info</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>flight_Info</td>
<td>Security</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.150</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>flight_Info</td>
<td>Security</td>
<td>flight_Info</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>flight_Info</td>
<td>Security</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>flight_info Probe</td>
<td>flight_Info</td>
<td>Reliability</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.013</td>
</tr>
<tr>
<td>flight_info Probe</td>
<td>flight_Info</td>
<td>Reliability</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.025</td>
</tr>
<tr>
<td>flight_info Probe</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.013</td>
</tr>
<tr>
<td>flight_info Probe</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.025</td>
</tr>
<tr>
<td>flight_info Probe</td>
<td>flight_Info</td>
<td>Security</td>
<td>book_flight</td>
<td>Response Time</td>
<td>0.013</td>
</tr>
<tr>
<td>flight_info Probe</td>
<td>flight_Info</td>
<td>Security</td>
<td>flight_Info</td>
<td>Response Time</td>
<td>0.025</td>
</tr>
<tr>
<td>hotel_info Interceptor</td>
<td>hotel_Info</td>
<td>Privacy</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.150</td>
</tr>
<tr>
<td>hotel_info Interceptor</td>
<td>hotel_Info</td>
<td>Privacy</td>
<td>hotel_Info</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Monitor</th>
<th>Service Monitored</th>
<th>Quality Type Monitored</th>
<th>Service Impacted</th>
<th>Quality Type Impacted</th>
<th>Impact Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>Privacy</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>privacyAct1</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.150</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>privacyAct1</td>
<td>hotel_Info</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>privacyAct1</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>Reliability</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.150</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>Reliability</td>
<td>hotel_Info</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>Reliability</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.150</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>hotel_Info</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>Security</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.150</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>Security</td>
<td>hotel_Info</td>
<td>Reliability</td>
<td>0.005</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>Security</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.250</td>
</tr>
<tr>
<td>Interceptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>Reliability</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.035</td>
</tr>
<tr>
<td>Probe</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hotel_info</td>
<td>hotel_Info</td>
<td>Reliability</td>
<td>hotel_Info</td>
<td>Response Time</td>
<td>0.070</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Monitor</th>
<th>Service Monitored</th>
<th>Quality Type Monitored</th>
<th>Service Impacted</th>
<th>Quality Type Impacted</th>
<th>Impact Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>hotel_info Probe</td>
<td>hotel_info</td>
<td>Response Time</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.035</td>
</tr>
<tr>
<td>hotel_info Probe</td>
<td>hotel_info</td>
<td>Response Time</td>
<td>hotel_info</td>
<td>Response Time</td>
<td>0.070</td>
</tr>
<tr>
<td>hotel_info Probe</td>
<td>hotel_info</td>
<td>Security</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>0.035</td>
</tr>
<tr>
<td>hotel_info Probe</td>
<td>hotel_info</td>
<td>Security</td>
<td>hotel_info</td>
<td>Response Time</td>
<td>0.070</td>
</tr>
</tbody>
</table>

Table 8.5: iTravel Monitoring Overhead Impacts

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Service Impacted</th>
<th>Quality Type Impacted</th>
<th>Impact Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>book_flight Interceptor</td>
<td>Response Time</td>
<td>book_flight</td>
<td>0.140</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>Response Time</td>
<td>book_hotel</td>
<td>0.125</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>Response Time</td>
<td>flight_info</td>
<td>0.180</td>
</tr>
<tr>
<td>hotel_info Interceptor</td>
<td>Response Time</td>
<td>hotel_info</td>
<td>0.140</td>
</tr>
</tbody>
</table>

These sets iMSI, iMO and iMC are used by the Impact Analyser to calculate the set of monitoring configurations with impacts for iTravel, iMCI. Too large to list here, iMCI maps each iMC that may be composed of any of the monitoring capabilities in Table 8.3 to a total set of monitoring impacts. This technique is described in detail in Chapter 5.

### 8.2 Implementation

We have implemented all aspects of the iTravel case study system, including the iTravel services and monitors, as well as our optimisation framework. Below, we describe the implementation of each aspect in detail.
8.2.1 iTravel Services and Monitors

The iTravel system has been implemented using Apache Tomcat/Axis on Ubuntu 9.04.

All web services including the flight and hotel info and booking services as well as the accounting and administration services are implemented in Perl, using the SOAP::LITE\(^2\) package for web services.

The probe-based monitors are also implemented in Perl, using the SOAP::LITE package for service consumption. As described in Chapter 2, the flight info Probe and book_flight Probe send service requests to the flight_info and book_flight services respectively, and measure the service responses for Security, Response Time and Reliability. The flight_info Interceptor and book_flight Interceptor measure the same quality types as the probes as well as Privacy, but are implemented as SOAP proxies that intercept and halt transmission of service requests and responses in order to measure each property. The monitors for the book_hotel and hotel_info services and the monitors Administration and Accounting subsystem are implemented in the same way as the Flight subsystem monitors.

The Generic Flight Eavesdropper and Generic Hotel Eavesdropper monitors are implemented using Wireshark to log all service requests and responses and each use these logs to measure the response time of service requests.

8.2.2 Design-Time Optimisation Framework

The Design-Time Optimisation Framework shown in Figure 5.2 was implemented as a set of Perl scripts that communicate via the file-system. The Requirements Analyser was not implemented, and instead the set of requirements was manually translated into a set of utility functions in a plain file, with format service name::quality type::monitoring level::quality level::utility level, with one utility value per line.

The Enumerator, Design-Time Impact Analyser and Optimiser were combined into a single Perl script that reads the set of utility values, a file of available monitors in the format monitor name::quality type::service name::monitoring level, and a file containing monitoring impacts in the format:

\[
\text{monitoring setting impacts:}
\]

\(^2\)http://www.soaplite.com

253
monitor name::quality type::service monitored::monitoring level::quality type impacted::service impacted::impact level
...
overheads:
service::quality type impacted::service impacted::impact level
...

The output from the Optimiser is a file containing the optimal monitoring configuration in the format monitor name::service name::monitoring level, with one line per monitoring capability. For the design-time optimisation, this configuration was manually applied to the set of iTravel monitors’ configuration files.

8.2.3 Run-Time Optimisation Framework

The Run-Time Optimisation Framework shown in Figure 6.2 was implemented as an extension of the Design-Time Optimisation Framework. Similarly, each component was implemented in Perl. The central component of the Run-Time Optimisation Framework is the Runtime Controller. This was implemented as a Daemon with a web service interface that allows for communication with the QoS Analyser, Configuration Manager, and system administrator. Methods provided to the system administrator include: methods to add, remove or modify monitors; methods to add, remove or modify monitoring impact knowledge; and methods to add, remove or modify utility functions. Each of these was executed manually to initialise the system and when changes were necessary (for example when run-time modifications were performed). The methods provided to the QoS Analyser were for sending QoS measurements, in the form start time::end time::service name::quality type::average quality level. The method provided to the Configuration Manager gave the current optimal monitoring configuration, in the same format as used in the Design-Time Optimisation Framework.

The Configuration Manager was also implemented with a web service interface. This allowed for the Runtime Controller to send requests for new monitoring configurations in the form monitor name::service::quality type::monitoring level. The
Configuration Manager then translates these to monitoring settings applicable for each Monitor and sends the configuration request to the appropriate Monitoring Manager. Translations were performed using a basic lookup table that translated between terms such as “Response Time” and “Speed”, and translated monitor names such as “flight_info Probe” to actual service addresses.

The Monitoring Managers were implemented as web services that were placed along with each monitor they were responsible for managing. Each Monitoring Manager provides a configure method with parameters for quality type, service and monitoring level. The Monitoring Manager then updates the configuration file for the corresponding Monitor with this new setting. Monitoring Managers also provide a method get_qos that takes parameters service, quality type, start time, and end time, and returns the average quality level for that quality of that service service over that time period, as measured by all available monitors.

The Run-Time Impact Analyser was also implemented as a Perl script (note that this is a separate component with different functionality to the Design-Time Impact Analyser). The script takes as input a file containing a set of QoS records in the format Monitoring Setting 1, Monitoring Setting 2,...Monitoring Setting N,Quality Type 1, Service 1, Measured Quality Level (that is, each line in the file corresponds to a monitoring configuration and set of QoS levels measured under that configuration). The Run-Time Impact Analyser then executes the algorithms described in Chapter 7 to calculate the impacts of each individual monitoring setting. These are output to a second file in the same format as used for input by the Optimiser (a file containing one monitoring setting impact per line and one monitoring overhead impact per line).

8.3 Initial Optimisation at Design Time

We have performed optimisation of the iTravel case study using the information on monitoring requirements and impacts, along with information recorded about the baseline response times for the iTravel Web Services from previous sections. Optimisation yielded 49 equally optimal monitoring configurations (i.e. configurations with the same utility). These configurations were typically equal to the optimal configuration, but with additional
monitoring enabled that provided no extra monitoring coverage but also did not cause a requirement to be breached (for example response time of a service may have increased from 2.6 to 2.8 seconds, which does not change the system utility). Therefore, the configuration with the lowest total response time was selected from the set of equally optimal configurations. In this case, the selected optimal configuration also had the lowest reliability for all services from the set of optimal configurations. When this is not the case, the system designer may select which quality type takes preference (including balancing between quality types) by adding requirements to the system with very low utility values.

This optimal monitoring configuration is shown in Table 8.6. Where Table 8.6 does not show a monitoring capability, that monitoring capability is set to a monitoring level of 0.

This optimal monitoring configuration includes using the admin Interceptor to monitor DLP and security of the admin service; the billing Probe to monitor security of the billing service; the book_flight, flight_info, book_hotel and hotel_info Probes to monitor the reliability and security of their target services; the book_flight, flight_info, book_hotel and hotel_info Interceptors to monitor their target services for DLP; the book_hotel Interceptor to monitor Privacy Act 1; each Eavesdropper monitor to measure its target service for response time; and the marketing Interceptor to monitor for the marketing service for DLP, Privacy Act 2, security and the Spam Act.

The resulting predicted QoS levels under this configuration are shown in Table 8.7, alongside the predicted QoS levels for the (default) maximum monitoring configuration. Table 8.7 shows that all requirements on DLP and security, as well as the Privacy Acts and Spam Act will be met, since they are monitored and no monitors impact them. Predicted reliability will meet all requirements under the optimal configuration, however none of the reliability requirements will be met under the max configuration. This is due to the reliability impacts of the Interceptor-based monitors. The response times under the optimal configuration vary, with no service meeting the strictest (2-second) requirement. However, the response time of the hotel_info service meets the 2.5-second requirement, whilst the remaining services all meet the 4-second requirement. Under the max configuration, the response times of all services are predicted to meet
<table>
<thead>
<tr>
<th>Monitor</th>
<th>Service</th>
<th>Quality Type</th>
<th>Monitoring Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin Interceptor</td>
<td>admin</td>
<td>DLP</td>
<td>1</td>
</tr>
<tr>
<td>admin Interceptor</td>
<td>admin</td>
<td>Security</td>
<td>1</td>
</tr>
<tr>
<td>billing Probe</td>
<td>bill_consumer</td>
<td>Security</td>
<td>1</td>
</tr>
<tr>
<td>book_flight Probe</td>
<td>book_flight</td>
<td>Reliability</td>
<td>1</td>
</tr>
<tr>
<td>book_flight Probe</td>
<td>book_flight</td>
<td>Security</td>
<td>1</td>
</tr>
<tr>
<td>book_flight Interceptor</td>
<td>book_flight</td>
<td>DLP</td>
<td>1</td>
</tr>
<tr>
<td>book_hotel Probe</td>
<td>book_hotel</td>
<td>Reliability</td>
<td>1</td>
</tr>
<tr>
<td>book_hotel Probe</td>
<td>book_hotel</td>
<td>Security</td>
<td>1</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>book_hotel</td>
<td>DLP</td>
<td>1</td>
</tr>
<tr>
<td>book_hotel Interceptor</td>
<td>book_hotel</td>
<td>Privacy Act 1</td>
<td>1</td>
</tr>
<tr>
<td>flight_info Probe</td>
<td>flight_info</td>
<td>Reliability</td>
<td>1</td>
</tr>
<tr>
<td>flight_info Probe</td>
<td>flight_info</td>
<td>Security</td>
<td>1</td>
</tr>
<tr>
<td>flight_info Interceptor</td>
<td>flight_info</td>
<td>DLP</td>
<td>1</td>
</tr>
<tr>
<td>Generic Flight Eavesdropper</td>
<td>book_flight</td>
<td>Response Time</td>
<td>1</td>
</tr>
<tr>
<td>Generic Flight Eavesdropper</td>
<td>flight_info</td>
<td>Response Time</td>
<td>1</td>
</tr>
<tr>
<td>Generic Hotel Eavesdropper</td>
<td>book_hotel</td>
<td>Response Time</td>
<td>1</td>
</tr>
<tr>
<td>Generic Hotel Eavesdropper</td>
<td>hotel_info</td>
<td>Response Time</td>
<td>1</td>
</tr>
<tr>
<td>hotel_info Probe</td>
<td>hotel_info</td>
<td>Reliability</td>
<td>1</td>
</tr>
<tr>
<td>hotel_info Probe</td>
<td>hotel_info</td>
<td>Security</td>
<td>1</td>
</tr>
<tr>
<td>hotel_info Interceptor</td>
<td>hotel_info</td>
<td>DLP</td>
<td>1</td>
</tr>
<tr>
<td>marketing Interceptor</td>
<td>marketing</td>
<td>DLP</td>
<td>1</td>
</tr>
<tr>
<td>marketing Interceptor</td>
<td>marketing</td>
<td>Privacy Act 2</td>
<td>1</td>
</tr>
<tr>
<td>marketing Interceptor</td>
<td>marketing</td>
<td>Security</td>
<td>1</td>
</tr>
<tr>
<td>marketing Interceptor</td>
<td>marketing</td>
<td>Spam Act</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8.6: iTravel Optimal Monitoring Configuration
only the 8 and 10 second requirements.

8.3.1 Design-Time Optimisation Evaluation

We have configured the case study system with both the optimal and max configurations above, and performed a series of experiments under varying load conditions to determine the resulting response times and reliability levels of each service, under each configuration.

Figure 8.2 shows the average response times for each service under each monitoring configuration. The average number of simultaneous clients per service was varied from 1 to $10^3$, and the response times were measured and averaged over a total of 10,000 service executions (1,000 per load level). Figure 8.2 shows that the average response time for each service under the max configuration was approximately twice as long the corresponding service under the optimal configuration. This reflects the predictions made in the design-time optimisation above.

Whilst the max monitoring configuration performs more monitoring than the optimal

\footnote{Whilst this figure seems low, 10 simultaneous clients per service equates to 50 simultaneous clients in total, and assuming that an active client sends a service request every two minutes on average and that the service takes 10 seconds to respond, this represents approximately 600 clients actively using the system at the same time.}
<table>
<thead>
<tr>
<th>Service</th>
<th>Quality Type</th>
<th>Optimal Quality Level</th>
<th>Max Quality Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin</td>
<td>DLP</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Admin</td>
<td>Security</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>billing</td>
<td>DLP</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>billing</td>
<td>Reliability</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>billing</td>
<td>Response Time (s)</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>billing</td>
<td>Security</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>book_flight</td>
<td>DLP</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>book_flight</td>
<td>Reliability</td>
<td>0.990</td>
<td>0.960</td>
</tr>
<tr>
<td>book_flight</td>
<td>Response Time (s)</td>
<td>3.370 (s)</td>
<td>6.740 (s)</td>
</tr>
<tr>
<td>book_flight</td>
<td>Security</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>book_hotel</td>
<td>DLP</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>book_hotel</td>
<td>Privacy Act 1</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>book_hotel</td>
<td>Reliability</td>
<td>0.990</td>
<td>0.960</td>
</tr>
<tr>
<td>book_hotel</td>
<td>Response Time (s)</td>
<td>3.970 (s)</td>
<td>7.160 (s)</td>
</tr>
<tr>
<td>book_hotel</td>
<td>Security</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>flight_info</td>
<td>DLP</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>flight_info</td>
<td>Reliability</td>
<td>0.990</td>
<td>0.960</td>
</tr>
<tr>
<td>flight_info</td>
<td>Response Time (s)</td>
<td>3.220 (s)</td>
<td>5.400 (s)</td>
</tr>
<tr>
<td>flight_info</td>
<td>Security</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>hotel_info</td>
<td>DLP</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>hotel_info</td>
<td>Reliability</td>
<td>0.980</td>
<td>0.960</td>
</tr>
<tr>
<td>hotel_info</td>
<td>Response Time (s)</td>
<td>2.470 (s)</td>
<td>4.274 (s)</td>
</tr>
<tr>
<td>hotel_info</td>
<td>Security</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Marketing</td>
<td>Security</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Marketing</td>
<td>DLP</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Marketing</td>
<td>Privacy Act 2</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Marketing</td>
<td>Spam Act</td>
<td>1.000</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Table 8.7: iTravel Optimal and Max Monitoring Configuration QoS Levels
Service | Optimal | Max  
--- | --- | ---  
book_flight | 0.99 | 0.96  
flight_info | 1.00 | 0.97  
book_hotel | 0.98 | 0.95  
hotel_info | 0.99 | 0.96  

Table 8.8: iTravel Case Study Run-Time Optimisation Reliability

configuration, it does not actually increase coverage (rather, it monitors the same Qualities of Service multiple times). As such, the difference in utility between the max and optimal monitoring configurations is related to their output QoS levels only. The QoS levels that vary in this case are the response time and reliability of book_flight, flight_info, book_hotel and hotel_info.

Figure 8.2 shows the average response times under each monitoring configuration. Both configurations provide response times under 10 seconds, meeting requirements SLA1, SLA2, SLA3 and SLA4. However, the response times of the max configuration are higher than those of the optimal configuration, and do not meet the business goal requirements G5 to G20 as frequently. The average value generated from business goals G5 to G20 across all services and load levels for the max configuration was $12,250, whilst the average for the optimal configuration was $15,875 - an increase of approximately 30%.

Table 8.8 shows the average reliability levels under each monitoring configuration. These values are an average reliability after each service was executed 1,000 times for each of the optimal and max monitoring configurations. The reliability for each service under the optimal configuration meets the business goal requirements G1, G2, G3 and G4 as well as the SLA requirements SLA5, SLA6, SLA7 and SLA8. This provides net value to iTravel of $65,000 per month. The reliability under the max configuration meets the business goal requirements G1 and G2, but not G3 and G4, and meets none of the SLA requirements. This provides net value to iTravel of $5,000 per month, which is $60,000 per month less than the optimal configuration.

Overall, the optimal configuration provided an average increase of business value of approximately $60,000 compared to the max configuration, as well as faster response times and greater reliability, whilst maintaining the same level of monitoring coverage.
8.4 Run-Time Optimisation

In this section, we evaluate the effectiveness of performing re-optimisation after a system modification as well as the effectiveness of continuous run-time re-optimisation. iTravel has added a new quality type, along with a new monitor to observe that quality type, monitoring impacts, and requirements on that quality type. These changes and their impacts are discussed below.

In order to increase marketing effectiveness, iTravel plans to monitor service executions for marketing analysis purposes. Therefore, iTravel now has a requirement to monitor the book_hotel, hotel_info, book_flight and flight_info services for marketing analysis. The value of performing this monitoring is $1,000 per service. This monitoring is performed by the existing Interceptor-based monitors. The impacts of performing this monitoring are measured to be a 25% monitoring setting impact on response time for the monitored service, and a 10% monitoring setting impact on response time for the collocated service, with no change to the overhead impacts of the Interceptor-based monitors. The iTravel system (including the requirements, monitors and monitoring impact knowledge) has been modified to include these changes.

As per the previous section, we have calculated the initial optimal monitoring configuration under the new conditions. This optimal monitoring configuration is the same as that in Table 8.6, except that each Interceptor monitor will now also monitor for marketing analysis at a monitoring level of 1.

We have applied and tested four monitoring configurations in this system:

(1) The previous optimal monitoring configuration that will not monitor for marketing analysis;

(2) The new optimal monitoring configuration that may monitor for marketing analysis;

(3) A continuously re-optimised monitoring configuration that will be re-optimised every 100 service executions; and

(4) The default, max monitoring configuration.
As per the previous experiments and QoS baseline measurements, we tested each configuration with load levels ranging from 1 to 10 simultaneous clients per service, and measured the resulting response times and reliability of each service under each load level and monitoring configuration.

Each configuration was executed for each load level 1,000 times. For the continuously optimal monitoring configuration, re-optimisation was performed every 100 service executions. This re-optimisation accounted for changes in response time due to load. This re-optimisation is important in this case, because as the system load increases, response times increase and the value of performing monitoring for marketing analysis becomes lower than its cost.

The results of executing the new iTravel system under the previously optimal configuration and the max configuration are shown in Figure 8.3.

Figure 8.3 shows the response times for each service under the previously optimal (Old Opt) and max (Max) configurations. These results are similar to those in Section 8.3.1, except that response times for the max configuration are all higher due to the extra monitoring. Since the new marketing analysis monitoring capabilities do not impact reliability, the differences in utility for all reliability-related requirements remain the same. However, as per the previous experiment, the response times under the max configuration do not meet as many requirements as frequently as those under the previously optimal configuration. The average value generated from business goals G5 to G20 across all services and load levels for the previously optimal configuration was $15,500, whilst the average for the max configuration was $11,625 - an increase of approximately 30%. Once again, both configurations provide response times under 10 seconds, meeting requirements SLA1, SLA2, SLA3 and SLA4.

The monitoring settings in the new optimal monitoring configuration differ from the previously optimal monitoring configuration in only one respect: the book_hotel service is monitored for marketing analysis by the book_hotel Interceptor. Figure 8.4 shows the results of these monitoring configurations. Since the new optimal monitoring configuration only impacts the response time of the book_hotel and hotel_info services, only this information is shown for comparison.

The new optimal configuration performs monitoring for marketing analysis on the
The book_flight service, increase the response time of the book_flight and flight_info services, as reflected in Figure 8.4. The average utility provided by business goals G5 to G12 (the requirements relating to response time of the book_flight and flight_info services) under the new optimal configuration was $7,750, compared to $8,250 under the old optimal configuration, a total difference of $500 or approximately 7%. However, since the book_flight service has been monitored for marketing analysis, there is an additional benefit of $1,000 under the new optimal configuration. This means that overall, the new optimal monitoring configuration provides $500 (approximately 7%) more value than the old optimal monitoring configuration.

We previously evaluated the effectiveness of performing once-off optimisation for system changes only. The continuously re-optimised configuration allows us to measure the effectiveness of performing continuous run-time optimisation, which accounts for short-to-medium term changes in system load. This allows for monitoring levels to be increased when the system is under lower load (and therefore response time-based requirements will still be met).

For these experiments, we executed the system described above, starting with the new optimal configuration and re-optimising every 100 client executions.

Figure 8.5 shows the results of this experiment. In Figure 8.5 the average response time for each service and load level is shown. The dotted, red points in the graph represent...
that particular service not being monitored for marketing analysis\textsuperscript{4}. Otherwise, the monitoring configuration was the same as the new optimal configuration from Section 8.4. The effects of re-optimisation are clearly visible in Figure 8.5. Since the value of monitoring for marketing analysis is $1,000 per service and the value of meeting any response time requirement for a service is at least $1,250, monitoring for marketing analysis only takes place when it will not cause a response time requirement to be breached that would have otherwise been met. For example, when all services produce response times well below 2 seconds, monitoring for marketing analysis is performed. However, as each service reaches near to a 2-second response time, monitoring for marketing analysis is disabled, to ensure that the response time requirements are met. This happens again for the 2.5-second requirement for all services except flight_info at the load level with 6 simultaneous clients, and for flight_info with 8 simultaneous clients. With 8 simultaneous clients, the book_hotel service is also not monitored. At 9, only the flight_info service is monitored, and at 10 all services except book_flight are monitored.

The average value for response time and marketing analysis provided under the continuously re-optimised configuration is $45,000. This comes from $14,000 value for meeting response time goals, and $31,000 value for meeting marketing analysis.

\textsuperscript{4}In each of these cases, marketing analysis was monitored for the first 100 service executions only.
goals. In comparison, the new optimal configuration provides a total of $35,000 value for both response time and marketing analysis ($10,000 for monitoring book_flight for marketing analysis and $25,000 for meeting response time goals). Figure 8.6 shows where this difference comes from. In Figure 8.6, the response times of the continuously re-optimised and new optimised configurations are shown together. The continuously re-optimised configuration managed to ensure no response time goals were breached that would not have already been breached due to load, whilst monitoring for marketing analysis where possible. Overall, continuously re-optimising the monitoring configuration increased the business value for response time and marketing analysis requirements by $10,000, or approximately 25% (once again, all other requirements remain unchanged).

8.5 Impact Analysis

The previous experiments in this chapter were performed on systems with known monitoring impacts. However, these impacts may not always be known or that knowledge may not always be up to date. Therefore, we have designed a set of experiments to test the effectiveness of our monitoring optimisation techniques on a system with some unknown monitoring impacts.

We have modified the system described in Section 8.4 by removing all knowledge of
the marketing analysis monitoring impacts on response time. We have also modified the monitors for marketing analysis to produce a 1% impact on the reliability of the service they monitor without providing this knowledge to the system. In total, we have four monitoring capabilities - monitoring each of the book_flight, flight_info, book_hotel and hotel_info services for marketing analysis using the Interceptor monitor of each of those services. We consider that each of these monitoring capabilities may impact the response time or reliability of any of the flight or hotel services. Therefore, we have 32 parameters (from 4 monitoring capabilities that may impact 2 quality types of 4 services) that must be determined by the impact analyser. Since the monitoring overhead for each Interceptor service is known, this knowledge (along with all other monitoring knowledge) is used by the impact analyser rather than being re-learnt. As such, this experiment represents the requirement to learn the impacts of a moderately-sized system change at run-time.

We re-executed the continuous run-time optimisation experiments above for this new system. Since the impacts of the new monitoring settings are unknown and not available through historical records, we executed the system whilst enabling and disabling the marketing analysis monitors for each of the book_flight, flight_info, book_hotel and hotel_info services. Since the value of marketing analysis is low (lower than the
## Executions

### Analysed Impacts

<table>
<thead>
<tr>
<th>#Executions</th>
<th>book_flight</th>
<th>flight_info</th>
<th>book_hotel</th>
<th>hotel_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.10</td>
<td>0.26</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>200</td>
<td>0.11</td>
<td>0.25</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>300</td>
<td>0.10</td>
<td>0.25</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>400</td>
<td>0.11</td>
<td>0.26</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>500</td>
<td>0.11</td>
<td>0.25</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>600</td>
<td>0.11</td>
<td>0.25</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>700</td>
<td>0.11</td>
<td>0.24</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>800</td>
<td>0.10</td>
<td>0.24</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>900</td>
<td>0.10</td>
<td>0.24</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1000</td>
<td>0.10</td>
<td>0.25</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 8.9: Response Time Impacts of Monitoring flight_info for Marketing Analysis in the iTravel Case Study

In order to learn the response time and reliability impacts of marketing analysis more quickly, if this assumption does not hold in a certain case, then marketing analysis impacts could be determined over a longer time by waiting for optimal configurations that provide the required information.

Monitoring for each service was enabled for 25 service executions before the next service was monitored, so that after 100 service executions, each service would have been monitored for 25 executions with between 1 and 10 simultaneous clients (as per previous experiments). The IMPACT ANALYSER was run every 100 service executions, and the results of impact analysis are shown in Tables 8.9 to 8.16.

Tables 8.9 to 8.12 show the response time impacts determined by the IMPACT ANALYSER. Each table shows the impacts of one monitoring capability. For example, Table 8.9 shows the response time impacts of monitoring flight_info for marketing analysis with the flight_info Interceptor monitor. The impacts were determined at between 100 and 1000 executions of the flight_info service. Here, 100 executions means that the
<table>
<thead>
<tr>
<th>#Executions</th>
<th>Analysed Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>book_flight</td>
</tr>
<tr>
<td>100</td>
<td>0.00</td>
</tr>
<tr>
<td>200</td>
<td>0.01</td>
</tr>
<tr>
<td>300</td>
<td>0.00</td>
</tr>
<tr>
<td>400</td>
<td>0.01</td>
</tr>
<tr>
<td>500</td>
<td>0.01</td>
</tr>
<tr>
<td>600</td>
<td>0.01</td>
</tr>
<tr>
<td>700</td>
<td>0.01</td>
</tr>
<tr>
<td>800</td>
<td>0.00</td>
</tr>
<tr>
<td>900</td>
<td>0.00</td>
</tr>
<tr>
<td>1000</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 8.10: Response Time Impacts of Monitoring hotel_info for Marketing Analysis in the iTravel Case Study

<table>
<thead>
<tr>
<th>#Executions</th>
<th>Analysed Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>book_flight</td>
</tr>
<tr>
<td>100</td>
<td>0.25</td>
</tr>
<tr>
<td>200</td>
<td>0.26</td>
</tr>
<tr>
<td>300</td>
<td>0.26</td>
</tr>
<tr>
<td>400</td>
<td>0.26</td>
</tr>
<tr>
<td>500</td>
<td>0.26</td>
</tr>
<tr>
<td>600</td>
<td>0.26</td>
</tr>
<tr>
<td>700</td>
<td>0.26</td>
</tr>
<tr>
<td>800</td>
<td>0.26</td>
</tr>
<tr>
<td>900</td>
<td>0.25</td>
</tr>
<tr>
<td>1000</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Table 8.11: Response Time Impacts of Monitoring book_flight for Marketing Analysis in the iTravel Case Study
# Executions

## Analysed Impacts

<table>
<thead>
<tr>
<th>#Executions</th>
<th>book_flight</th>
<th>flight_info</th>
<th>book_hotel</th>
<th>hotel_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.00</td>
<td>0.00</td>
<td>0.26</td>
<td>0.11</td>
</tr>
<tr>
<td>200</td>
<td>0.01</td>
<td>0.00</td>
<td>0.26</td>
<td>0.11</td>
</tr>
<tr>
<td>300</td>
<td>0.00</td>
<td>0.00</td>
<td>0.25</td>
<td>0.10</td>
</tr>
<tr>
<td>400</td>
<td>0.01</td>
<td>0.00</td>
<td>0.25</td>
<td>0.10</td>
</tr>
<tr>
<td>500</td>
<td>0.01</td>
<td>0.00</td>
<td>0.24</td>
<td>0.10</td>
</tr>
<tr>
<td>600</td>
<td>0.01</td>
<td>0.00</td>
<td>0.24</td>
<td>0.10</td>
</tr>
<tr>
<td>700</td>
<td>0.01</td>
<td>0.00</td>
<td>0.24</td>
<td>0.10</td>
</tr>
<tr>
<td>800</td>
<td>0.00</td>
<td>0.00</td>
<td>0.24</td>
<td>0.10</td>
</tr>
<tr>
<td>900</td>
<td>0.00</td>
<td>0.00</td>
<td>0.24</td>
<td>0.10</td>
</tr>
<tr>
<td>1000</td>
<td>0.00</td>
<td>0.00</td>
<td>0.24</td>
<td>0.10</td>
</tr>
</tbody>
</table>

Table 8.12: Response Time Impacts of Monitoring book_hotel for Marketing Analysis in the iTavel Case Study

Service was executed 100 times with monitoring enabled. The four “Analysed Impacts” columns show the results of the Impact Analyser for each number of executions. For example, the first row (with 100 executions) shows that the impact on book_flight is 10%, flight_info is 26%, book_hotel is 1% and hotel_info is 1%. In this case, the first impact is correct, whilst the last three are incorrect by 1% (the correct impacts as determined through benchmarking but not provided to the system are 10%, 25%, 0% and 0%, respectively and we assume that impacts are independent). As the number of executions increases, the accuracy of the determined impacts tends to increase. By 1000 executions, the results of the Impact Analyser (the last row in Table 8.9) have no error. However, this was not always the case, as each of the hotel_info and book_hotel impact analyses show one error of 1% (In Table 8.10 the impact on book_hotel in last row should be 0.10 rather than 0.09 and in Table 8.12 the impact on book_hotel in last row should be 0.25 rather than 0.24).

Over all, we were able to calculate the impact of each of the four monitoring capabilities on response time to within 1% after 1000 executions. The average error after 1000 executions was 1.4% (due to an average error of 0.5% over the total error of 35%,
Table 8.13: Reliability Impacts of Monitoring flight\_info for Marketing Analysis in the iTravel Case Study

<table>
<thead>
<tr>
<th>#Executions</th>
<th>book_flight</th>
<th>flight_info</th>
<th>book_hotel</th>
<th>hotel_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.000</td>
<td>0.000</td>
<td>0.030</td>
<td>0.020</td>
</tr>
<tr>
<td>200</td>
<td>0.000</td>
<td>0.010</td>
<td>0.005</td>
<td>0.005</td>
</tr>
<tr>
<td>300</td>
<td>0.000</td>
<td>0.013</td>
<td>0.003</td>
<td>0.000</td>
</tr>
<tr>
<td>400</td>
<td>0.000</td>
<td>0.015</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>500</td>
<td>0.000</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
</tr>
<tr>
<td>600</td>
<td>0.000</td>
<td>0.018</td>
<td>0.013</td>
<td>0.000</td>
</tr>
<tr>
<td>700</td>
<td>0.000</td>
<td>0.019</td>
<td>0.013</td>
<td>0.000</td>
</tr>
<tr>
<td>800</td>
<td>0.000</td>
<td>0.017</td>
<td>0.010</td>
<td>0.000</td>
</tr>
<tr>
<td>900</td>
<td>0.000</td>
<td>0.022</td>
<td>0.008</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.000</td>
<td>0.020</td>
<td>0.008</td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>0.000</td>
<td>0.013</td>
<td>0.004</td>
<td>0.001</td>
</tr>
<tr>
<td>3000</td>
<td>0.000</td>
<td>0.012</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>4000</td>
<td>0.000</td>
<td>0.012</td>
<td>0.001</td>
<td>0.002</td>
</tr>
<tr>
<td>5000</td>
<td>0.000</td>
<td>0.013</td>
<td>0.000</td>
<td>0.002</td>
</tr>
</tbody>
</table>

for each monitoring capability). Through analysis, we have determined that using these impacts with their associated errors would not have changed the outcome of the experiments in Section 8.4 (i.e., run-time optimisation would have led to the same monitoring configurations being used even with the 1.4% error in impact knowledge).

Tables 8.13 to 8.16 show the reliability impacts determined by the Impact Analyser. As per Tables 8.9 to 8.12, each table shows the impacts of one monitoring capability. For example, Table 8.13 shows the reliability impacts of monitoring flight\_info for marketing analysis with the flight\_info Interceptor. In this case, the real impact is 1% (0.01) on flight\_info and 0% on all other services.

Impact analysis on reliability required more service executions than it did for response time to determine accurate results. For example, Table 8.13 shows that even after 5000 service executions, the analysed impact for flight\_info was 1.3% (it should
Table 8.14: Reliability Impacts of Monitoring hotel_info for Marketing Analysis in the iTravel Case Study

<table>
<thead>
<tr>
<th>#Executions</th>
<th>Analysed Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>book_flight</td>
</tr>
<tr>
<td>100</td>
<td>0.010</td>
</tr>
<tr>
<td>200</td>
<td>0.015</td>
</tr>
<tr>
<td>300</td>
<td>0.003</td>
</tr>
<tr>
<td>400</td>
<td>0.000</td>
</tr>
<tr>
<td>500</td>
<td>0.000</td>
</tr>
<tr>
<td>600</td>
<td>0.000</td>
</tr>
<tr>
<td>700</td>
<td>0.000</td>
</tr>
<tr>
<td>800</td>
<td>0.000</td>
</tr>
<tr>
<td>900</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>0.000</td>
</tr>
<tr>
<td>3000</td>
<td>0.000</td>
</tr>
<tr>
<td>4000</td>
<td>0.000</td>
</tr>
<tr>
<td>5000</td>
<td>0.000</td>
</tr>
<tr>
<td>#Executions</td>
<td>Analysed Impacts</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------</td>
</tr>
<tr>
<td></td>
<td>book_flight</td>
</tr>
<tr>
<td>100</td>
<td>0.000</td>
</tr>
<tr>
<td>200</td>
<td>0.010</td>
</tr>
<tr>
<td>300</td>
<td>0.013</td>
</tr>
<tr>
<td>400</td>
<td>0.015</td>
</tr>
<tr>
<td>500</td>
<td>0.012</td>
</tr>
<tr>
<td>600</td>
<td>0.008</td>
</tr>
<tr>
<td>700</td>
<td>0.013</td>
</tr>
<tr>
<td>800</td>
<td>0.012</td>
</tr>
<tr>
<td>900</td>
<td>0.011</td>
</tr>
<tr>
<td>1000</td>
<td>0.012</td>
</tr>
<tr>
<td>2000</td>
<td>0.011</td>
</tr>
<tr>
<td>3000</td>
<td>0.010</td>
</tr>
<tr>
<td>4000</td>
<td>0.009</td>
</tr>
<tr>
<td>5000</td>
<td>0.009</td>
</tr>
</tbody>
</table>

Table 8.15: Reliability Impacts of Monitoring book_flight for Marketing Analysis in the iTravel Case Study
### Table 8.16: Reliability Impacts of Monitoring book_hotel for Marketing Analysis in the iTravel Case Study

<table>
<thead>
<tr>
<th>#Executions</th>
<th>book_flight</th>
<th>flight_info</th>
<th>book_hotel</th>
<th>hotel_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.020</td>
</tr>
<tr>
<td>200</td>
<td>0.000</td>
<td>0.000</td>
<td>0.005</td>
<td>0.020</td>
</tr>
<tr>
<td>300</td>
<td>0.000</td>
<td>0.000</td>
<td>0.007</td>
<td>0.013</td>
</tr>
<tr>
<td>400</td>
<td>0.000</td>
<td>0.000</td>
<td>0.010</td>
<td>0.020</td>
</tr>
<tr>
<td>500</td>
<td>0.000</td>
<td>0.000</td>
<td>0.010</td>
<td>0.018</td>
</tr>
<tr>
<td>600</td>
<td>0.000</td>
<td>0.000</td>
<td>0.017</td>
<td>0.020</td>
</tr>
<tr>
<td>700</td>
<td>0.000</td>
<td>0.000</td>
<td>0.014</td>
<td>0.019</td>
</tr>
<tr>
<td>800</td>
<td>0.001</td>
<td>0.000</td>
<td>0.017</td>
<td>0.017</td>
</tr>
<tr>
<td>900</td>
<td>0.004</td>
<td>0.000</td>
<td>0.013</td>
<td>0.013</td>
</tr>
<tr>
<td>1000</td>
<td>0.006</td>
<td>0.000</td>
<td>0.013</td>
<td>0.014</td>
</tr>
<tr>
<td>2000</td>
<td>0.001</td>
<td>0.000</td>
<td>0.013</td>
<td>0.007</td>
</tr>
<tr>
<td>3000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.015</td>
<td>0.004</td>
</tr>
<tr>
<td>4000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.014</td>
<td>0.002</td>
</tr>
<tr>
<td>5000</td>
<td>0.002</td>
<td>0.000</td>
<td>0.013</td>
<td>0.001</td>
</tr>
</tbody>
</table>
have been 1.0 and so was incorrect by 0.3%) and the analysed impact for **hotel_info** was 0.2% (it should have been 0 and so was incorrect by 0.2%). This inaccuracy is due to two factors. Firstly, the true impact on **reliability** is only 1%, meaning that analysis must be accurate to within 0.1% in order to have less than 10% error (compare this to **response time** impacts, the smallest of which is 10%). Secondly, **response times** can be more effectively averaged from smaller samples, whereas **reliability** requires more samples because results are binary.

Tables 8.13 to 8.16 show that after 5000 service executions, analysed impacts were incorrect by an average of 15% (from a total of 2.4% error from 16 parameters). This is due to the low impact size (1%) compared to the sample sizes (number of service executions) used. In order to decrease this error, more samples would be required. However, this error rate may be acceptable in many cases, provided the magnitude of the error is lower than the step size in requirements. In our example, requirements on **reliability** have a resolution of 1%, which means that an error in impact analysis of 0.15% will not affect the result of optimisation.

We have demonstrated that the **Impact Analyser** is capable of accurately determining monitoring impacts. The accuracy of analysis depends on the number of service executions as well as the quality types impacted.

### 8.6 Summary

We have implemented a case study system and used it to evaluate each of our web service monitoring configuration optimisation techniques, including the **Design-Time Optimiser**, the **Run-Time Optimiser** both once-off and continuously, and the **Impact Analyser**.

Performing design-time optimisation increased business value by $60,000 per month when compared to the default maximum monitoring configuration.

Performing a secondary (run-time) optimisation after system changes increased business value by $500 over the previous optimal configuration. However, extending this optimisation to be performed continuously at run-time increased business value by a further $9,250 of business value per month, when compared to the initial optimisation.

Impact analysis was performed on four monitors in the system in order to measure the
response time and reliability impacts of those monitors. The average error rate for measuring response time was 1.4% after 1000 service executions, and the average error rate for measuring reliability was 15% after 5000 service executions.
Chapter 9

Conclusions

In this thesis, we identified the need to optimise the configuration of Web Service monitoring systems by balancing the monitoring impacts and monitoring benefits of those systems. This involved first creating a classification of Web Service monitors and identifying the impacts that each monitor class is likely to have. We used this information when designing our Web Service monitoring configuration optimisation techniques, which were applied at both design-time and run-time. These optimisation techniques were extended with a method to automatically calculate the impact of monitors by performing regression analysis on monitoring logs. Finally, we measured the effectiveness of all of our approaches using a detailed case study.

Below, we summarise the contributions of this thesis and highlight potential future work for this research.

9.1 Research Contributions

In this thesis, we have described Service Oriented Architecture (SOA) and Web Service (WS)-based systems with particular emphasis on the need to measure the Quality of Service (QoS) of WS-based systems through monitoring. Whilst this monitoring was identified as being beneficial to a WS provider, the monitoring was expected to reduce the delivered QoS levels due to impacts on services being monitored. Therefore, we have identified the requirement to balance the needs of monitoring a WS-system (including meeting legal accountability requirements and measuring delivered QoS levels to validate
Service Level Agreements (SLAs)) with the needs of meeting goals for delivered quality levels.

**Service Monitoring Requirements** We used a realistic, fictional business scenario (iTravel) to demonstrate the needs of a web service provider with respect to web service monitoring optimisation. The key requirements that were identified were that:

- The service provider should have a generic profile of monitoring impacts;
- The service provider should be able to optimise the configuration of their web service monitoring system by balancing monitoring coverage with monitoring impacts at design-time;
- The service provider should be able to maintain an optimal monitoring configuration at run-time by continuously re-optimising their monitoring configuration based on changes to long-term load, new or modified services or monitors, changes to requirements, and system changes that affect baseline qualities of service or monitoring impact levels; and
- The service provider should be able to learn the impacts of monitors by analysing monitoring logs.

These requirements have been used to direct our research.

**Literature Review** Based on our research requirements, we have performed a survey of monitoring literature. This survey placed particular emphasis on monitoring for web services or distributed systems, and dynamic or adaptive monitoring systems, or those systems that take monitoring impact into account. Of the 38 works surveyed, 14 mentioned that minimising the impact of monitoring (whether it be performance, setup cost, or running cost) would be beneficial. Of those, eight had measured the impact of their monitoring solution. Whilst many works surveyed identified the need to minimise the performance impact or other costs of monitoring, few actually implemented techniques to achieve this, and fewer still benchmarked their results. Only eight of the works reviewed considered a trade-off between monitoring benefits and costs, and these were either manual
solutions or generic techniques. This highlighted that whilst minimising the impact of monitoring was considered valuable, most authors had not described methods to achieve this or even measured the impacts of their monitoring systems. None of the works reviewed provided a mechanism to automatically balance the benefits and costs of monitoring at run-time in terms of SLAs. Furthermore, those works that did consider the costs of monitoring either assumed monitoring costs were static and could be manually provided, or measured costs in terms of raw resource use of monitors, rather than linking costs to delivered QoS.

Based on our literature review and case study requirements, we identified the following set of major unsolved research requirements for WS monitoring:

1. Providing a generic classification of WS monitors and measuring the performance impact profiles of each monitoring class;
2. Providing a method to calculate an optimal monitoring configuration, based on costs due to the QoS impacts of monitoring and benefits from SLAs, business rules, and other sources;
3. Providing a method to maintain an optimal monitoring configuration at run-time, as system requirements, cost and benefits, and delivered QoS change; and
4. Providing a method to measure and continuously refine knowledge of monitoring impacts on a run-time system.

**Impacts of Web Service Monitors** We have identified no work that would meet the first research requirement identified above by quantifying the impact of various types of Web Services monitoring systems on the quality of service provided by those Web Services. To fill this gap, we have conducted a series of experiments in order to quantify and assess the impact of monitoring on Web Services in typical Web Service scenarios. We used response time as a target QoS to measure for impacts. These experiments provided three key results: firstly, that it is possible to eavesdrop on a Web Service with no measurable negative impact on response time. Secondly, that a blocking monitor may significantly increase the response time of a Web Service when compared with a non-blocking monitor. Thirdly, that the location of an interceptor monitor has an impact on the response
time of the Web Service being monitored. These results provided the rationale to investigate strategies to optimise a Web Service monitoring system at design-time and run-time.

**Design-Time Monitoring Optimisation** To meet the second research requirement identified above we have developed a **Design-Time Optimiser**, consisting of a framework and techniques for optimising the configuration of Web Service monitors in order to maximise the utility for a Web Services provider. These techniques were used to translate requirements from sources such as SLAs into a set of utility functions that were used to score the business value that would be achieved under each possible web service monitoring configuration. We have introduced a set of tools for generating randomised optimisation problems that, together with our iTravel case study, were used to measure the effectiveness of our technique over a wide range of possible application scenarios.

Our optimisation framework was used for a series of experiments that have demonstrated that both the system utility and performance can be improved by optimally configuring a Web Services monitoring system. Optimisation reduced the average response time of the iTravel scenario by approximately 30% from maximum monitoring, and reduced the average response time impact of monitoring by over 80% (the optimal monitoring configuration had an average 7% impact, versus the default maximum monitoring configuration’s 43%). For the iTravel scenario, the system with an optimised monitoring configuration dealt with twice as much load as the un-optimised system before a penalty for slow response times would have been paid (this was due to significant impacts on response time from duplicated monitoring in this scenario). To verify that the values selected for our motivating scenario were realistic, we repeated all tests with randomly generated utility functions and monitoring impacts under various load levels. The response times for the optimised solutions were on average 40% lower than maximum monitoring, and the optimised response time impacts were on average 70% lower than maximum monitoring. Utility values for optimised systems were 30% higher than utility values for systems with maximum monitoring, on average. The results of our experiments have demonstrated that both the utility and performance of a Web Services monitoring system can be increased by optimising the configuration of Web Service monitors.
Run-Time Monitoring Optimisation  To meet the third research requirement identified above, we have developed a Run-Time Optimiser, consisting of a framework and technique for run-time management and optimisation of a WS monitoring system. This framework has provided the capability to continuously monitor a WS monitoring system at run-time and perform optimisation as changes occur. We have also presented a heuristic optimisation routine that was used to efficiently solve the run-time optimisation problem. We have demonstrated that the use of Genetic Algorithm (GA)-based optimisation with search space pruning allows for systems with up to $10^{160}$ possible monitoring configurations (allowing for a system with approximately 500 monitoring capabilities) to be optimised within one hour, compared to the enumerative approach which will only solve for systems with up to approximately $10^6$ possible monitoring configurations (allowing for a system with approximately 20 monitoring capabilities) within this time-frame. Furthermore, the GA-based solver found optimal solutions for all experiments that could be solved by the enumerative technique. The run-time optimisation system was implemented and tested and achieved an average of a further 20% increase in utility across a wide range of randomly generated target systems when compared to once-off design-time optimisation.

Analysis of Monitoring Impacts  To meet the fourth research requirement identified above, we have developed an Impact Analyser, consisting of a technique for using run-time logs from a suite of Web Service monitors to determine the impact of individual monitors on any delivered QoS, according to the actions that each monitor is performing. This technique analyses logs that provide detail of a monitoring configuration for the system along with the QoSs that were delivered under that configuration. By comparing the delivered QoS between different monitoring configurations, we determine the impact levels of each monitor. These impact levels included overhead impact for a monitor and a set of monitoring setting impacts that describe the impacts of each individual monitoring action. We have described a case study system and generated a set of monitoring logs for that system, which were varied in size and noise level. These logs were then analysed to determine the impacts of monitors and monitoring settings in
the system. Overall our results have demonstrated that it is possible to determine the impacts of individual monitors from the integrated impacts of a set of monitors in a Web Service system, using only existing monitoring logs from that Web Service system. In general, having a large enough set of monitoring records with a range of monitoring configurations allows for monitoring impacts to be determined with a high degree of accuracy (an error level lower than the system’s natural noise level). If there are not enough monitoring configurations used then analysis may require manually configuring systems for the purpose of measuring monitoring impacts, provided the costs of modifying the monitoring configuration for the sole purpose of impact analysis are not too high.

**Case Study and Evaluation**  We have implemented a larger case study system and used it to further evaluate each of our web service monitoring configuration optimisation techniques, including the **Design-Time Optimiser**, the **Run-Time Optimiser**, and the **Impact Analyser**.

For the iTravel case study, the quality types that were impacted were reliability and response time. The total value of requirements on these quality types was $90,000 per month (i.e., this is the business value received for reliability and response time based requirements when all monitoring requirements and quality level requirements are met). The default monitoring configuration provided a total of $17,500 business value per month for the reliability and response time requirements. Performing design-time optimisation increased business value by $60,000 when compared to the default maximum monitoring configuration (yielding a total of $77,500 per month).

Performing a secondary (run-time) optimisation after the system was modified has increased business value by $500 over the previous optimal configuration. Extending this optimisation to be performed continuously at run-time has increased business value by a further $9,250 per month, when compared to the initial design-time optimisation.

Impact analysis was performed on four monitors in the system in order to measure the response time and reliability impacts of those monitors. The average error rate for measuring response time was 1.4% after 1000 service executions, and the average error rate for measuring reliability was 15% after 5000 service executions.
9.2 Future Work

We plan to extend the research described in this thesis in the following major directions: extended impact analysis, automatic translation of Web Service Level Agreement (WSLA)-format files into sets of utility functions, extension of the search space division techniques, investigation of alternate heuristic algorithms for run-time optimisation, extension of the run-time optimisation controller, extension of the impact analysis technique, and consideration of system adaptation beyond monitoring configuration.

First, we plan to investigate the impacts of common commercially available Web Service monitoring systems. This will allow us to both demonstrate the effectiveness of optimisation using these systems as well as benchmark alternate but functionally equivalent Web Service monitoring systems. Providing this information will help designers and engineers to select monitoring systems that have the least impact on the most valuable QoS aspects in their systems.

Our optimisation technique currently requires manual translation of requirements from SLAs into sets of utility functions. Since this is the only non-automated aspect of our system, it would be beneficial to provide a technique to automatically translate SLAs into utility functions. We plan to provide a method for automatic translation of sets of requirements in standard formats such as WSLA into utility functions that can be used in our system. This would allow for faster, less error-prone translation, and would help the run-time optimisation system to maintain an optimal state by continuously updating the set of utility functions as WSLAs change, rather than having to wait for manual updates after those changes.

We plan to extend our search-space division and pruning techniques by relaxing their constraints for no overlapping monitoring capabilities or monitoring impacts. This would allow for a trade-off between faster optimisation and the accuracy of that optimisation. For example, consider two monitors each with five identical monitoring capabilities. If all except one of the impacts of one monitor’s capabilities are higher than the other, we may prune that monitor under the assumption that it will probably not be part of an optimal solution. Evaluation of this technique would be required in order to determine the typical ideal balance between faster optimisation and the accuracy of that optimisation.

Our qualitative analysis has identified that genetic algorithms were suitable for run-
time optimisation, and indeed our experiments have shown that the GA-based heuristic optimisation algorithm has met our requirements. However, other classes of heuristic algorithms such as simulated annealing and swarm or particle-based algorithms may perform equally well or even better. Further quantitative experiments and analysis can be performed to evaluate these other heuristic algorithms with the aim to increase confidence in or accuracy of the optimisation.

The run-time optimisation controller currently re-optimises the system whenever there is either a system change (e.g. a new monitor or service is added) or there is a ‘significant’ change in QoS levels delivered. The significance of this change is currently defined by the user as a percentage difference in any QoS level delivered. Whilst these re-optimisation triggers were satisfactory for our case study, they may lead to unnecessary reconfiguration fluctuations in some cases. Therefore, we plan to design and evaluate methods for automatically dampening the run-time optimisation system that would reduce unnecessary fluctuations whilst maintaining a close-to-optimal monitoring configuration.

Our impact analysis technique determines the impacts of Web Service monitors only. We plan to extend this to determine the impacts of non-monitor elements or aspects of Web Service systems such as network configurations and the allocation levels of hardware resources to Web Services and their monitors. This would allow for more accurate impact analysis as well as the potential to provide information for the optimisation of resource allocation and network configurations in Web Service systems.

We have focused on the optimisation of the monitoring system in this research. A related issue is that when monitoring identifies issues with the services, adaptations to these services may need to be carried out. For example, certain quality requirements may not be met by certain services or a service composition, and changes may need to be made to the services or service composition. We plan to investigate approaches to run-time service adaptation such as service selection and re-composition that may be coordinated or merged with our monitoring optimisation framework.
Bibliography


[62] James Skene, Allan Skene, Jason Crampton, and Wolfgang Emmerich. The mon-
itorability of service-level agreements for application-service provision. In WOSP
'07: Proceedings of the 6th international workshop on Software and performance,
//doi.acm.org/10.1145/1216993.1216997.

[63] Tim Stevens, Joachim Vermeir, Marc De Leenheer, Chris Develder, Filip De Turck,
Bart Dhoedt, and Piet Demeester. Distributed service provisioning using stateful
anycast communications. Local Computer Networks, Annual IEEE Conference on,
0:165–174, 2007. ISSN 0742-1303. doi: http://doi.ieeecomputersociety.org/10.1109/

[64] Hong-Linh Truong, Robert Samborski, and Thomas Fahringer. Towards a frame-
work for monitoring and analyzing qos metrics of grid services. In E-SCIENCE
'06: Proceedings of the Second IEEE International Conference on e-Science and Grid
0-7695-2734-5. doi: http://dx.doi.org/10.1109/E-SCIENCE.2006.142.

University, March 2005.

[66] Liangzhao Zeng, Hui Lei, and Henry Chang. Monitoring the qos for web services.
In Proceedings of the 5th international conference on Service-Oriented Computing,
3-540-74973-8. doi: http://dx.doi.org/10.1007/978-3-540-74974-5_11. URL http:
//dx.doi.org/10.1007/978-3-540-74974-5_11.

[67] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A regression-based analytic
model for dynamic resource provisioning of multi-tier applications. In Proceedings of
the Fourth International Conference on Autonomic Computing, pages 27–40, Wash-

[68] Yue Zhang, Mark Panahi, and Kwei-Jay Lin. Service process composition with
qos and monitoring agent cost parameters. In 2008 10th IEEE Conference on E-
Appendices
Appendix A

Survey of Service Level Agreements

This section provides a summary of publicly available Service Level Agreements (SLAs) for major Web Service (WS) and cloud providers.

A.1 Aplicor

Aplicor provides Customer Relationship Management (CRM) and Enterprise Resource Planning (ERP) solutions via a Software as a Service (SaaS) platform, with the following guarantees[5]:

1. Availability of 99.5% per calendar month.

2. Latency to selected tier 1 Points-of-Presence’ of 50ms for North America, 90ms for Europe, Middle East and Asia, and 130ms for Asia Pacific.

3. Less than 0.1% packet loss.

4. Less than 250 microseconds jitter with a maximum of 10 milliseconds of jitter 99% of the time, per calendar month.

These Service Level Objectives (SLOs) are offered with a ‘Financial Guarantee’, the details of which are not publicly available.
A.2 Boomi

Boomi provides a SaaS platform hosting their own and third party services, and provides integration of services. Boomi provides an SLA covering Service Availability, with the following grades and penalties[14]:

- For Availability of less than 99.99%, a **Penalty** of 10% of the monthly account fee;
- For Availability of less than 99%, a **Penalty** of 20% of the monthly account fee;
- For Availability of less than 97%, a **Penalty** of 35% of the monthly account fee;
- For Availability of less than 95%, a **Penalty** of 50% of the monthly account fee; and
- For Availability of less than 90%, a **Penalty** of 100% of the monthly account fee.

Availability is measured per calendar month, and excludes scheduled and emergency maintenance.

A.3 Google Apps

Google provides ‘Google Apps Premier Edition’, which, whilst not implemented as Web Services, is a popular platform that offers applications as services over the Internet. Google Apps Premier Edition provides Google applications such as email, calendar, instant messaging, and word processing, with the following SLO and attached penalties based on service uptime[23]. The SLO is graded, with penalties scaled according to the level of service provided:

- For less than **95% Uptime**, 15 days worth of account fees are refunded;
- For between **95% and 99% Uptime**, 7 days worth account fees are refunded; and
- For between **99% and 99.9% Uptime**, 3 days worth of account fees are refunded.

Google defines downtime as 10 consecutive minutes of greater than 5% error rate, measured at the server side, excluding scheduled maintenance.
A.4 NetSuite

NetSuite provides business services such as CRM and business management via a SaaS solution. NetSuite also provides Suiteapp, an service directory for third party services. NetSuite offers the following quality and functionality agreements to customers[40]:

(1) A Warranty of Functionality, which guarantees that NetSuite services provide functionality described in user guides. The Warranty of Functionality does not guarantee lack of errors and faults, however. The Penalty for breaching this SLO is a Pro-rata refund of fees, and ability for consumers to terminate their contract.

(2) A Service Level Warranty, which guarantees an uptime level of 99.5% per month, excluding scheduled maintenance. The Penalty for breaching this SLO is a refund of one month’s fee.

(3) A Security, Data Integrity, and Availability Warranty, which provides security, data integrity and availability with ‘industry standard security’ using ‘commercially reasonable efforts’. The Penalty for breaching this SLO is a refund of 5 years of licence fees, and eligibility for early termination with a pro-rata refund.

(4) A Virus Warranty, which warrants the system to be free from viruses, worms, and malware. The Penalty for breaching this SLO is not reported by NetSuite.

NetSuite cap the penalty payable for any set of SLO breaches to 5 years of licence fees.

A.5 RightNow

RightNow provides a suite of CRM services via a SaaS model. RightNow provides three levels of SLAs[54]. A ‘Basic’ level offers a target of 99.5% Uptime, with no Penalty, measured at 15 minute intervals. A ‘Preferred’ level offers a target of 99.9% Uptime, with a Penalty to be paid for Uptime below 99.5%, measured at 5 minute intervals. A ‘Premier’ level offers a target of 99.95% Uptime, with a Penalty to be paid for Uptime below 99.9%, measured at 1 minute intervals. The penalties prescribed are listed as ‘Financial Credit’, but further details are not publicly available.
A.6 Salesforce

Salesforce is a primarily an online provider of CRM solutions. These solutions are provided through Web Services and other means such as the Force.com platform. Salesforce also provides a service directory where third party developers publish and sell their web applications. Whilst being a major provider of SaaS, Salesforce does not publicly offer any SLAs.

A.7 SAP

SAP provides variants of SaaS under different marketing strategies and names. SAP currently offers ‘Business ByDesign’, a business management package aimed at small-to-medium enterprises. SAP does not publicly offer any SLAs.

A.8 Windows Azure

Windows Azure provides a cloud computing platform for hosting .NET services, computation, data, and databases. There are separate SLAs for each of these aspects of the system[37].

The Azure ‘Storage’ SLA provides a guarantee of Functional Correctness, with two grades. For between 99% and 99.9% Correctness, a penalty of 10% of the monthly account fee is paid. For below 99% Correctness, a penalty of 25% of the monthly account fee is paid. Correctness is measured with respect to faults on data operations, such as adding, updating, and reading data, and includes a guarantee that the service will have 100% uptime.

The Azure ‘Compute’ and ‘.NET Service Hosting’ SLA provides a guarantee of Availability, with two grades. For between 99% and 99.95% Availability, a Penalty of 10% monthly account fee is paid. For less than 99% Availability, a Penalty of 25% monthly account fee is paid. Availability is measured every 2 minutes, and is defined as services communicating and being functionally correct.

The ‘Azure SQL’ SLA provides an Uptime guarantee, with two grades. For between 99% 99.9% Uptime, a Penalty of 10% of the monthly account fee is paid. For less
than 99% **Uptime**, a penalty of 25% of the monthly account fee is paid. Uptime is measured per calendar month, at 5 minute intervals.

### A.9 Summary

This survey of SLAs from major Web Service and cloud providers shows the breadth of SLAs that are offered. SLAs range from complex, multi-level agreements with graded consumer classes, to simple SLAs that guarantee uptime, with some providers offering no SLA at all.
Appendix B

Reference Guide to Formal Definitions

\( q \in Q \)

Q is set of all qualities, e.g. \{Response Time, Privacy, Reliability Correctness\}

\( s \in S \)

S is set of all services, e.g. \{book_flight, flight_info\}

\( m \in M \)

M is set of all monitors, e.g. \{book_flight Probe, book_flight Interceptor, Generic Eavesdropper, flight_info Probe, flight_info Interceptor\}

\( mcap = (m, s, q) \)

\( mcap \) represents a monitoring capability. A monitoring capability is a three-tuple consisting of a monitor, a service and a quality type, that describes the ability for that monitor to observe that quality type of that service. For example, (Generic Eavesdropper, book_flight, Response Time) is one monitoring capability.
ML is a set of monitoring levels that each monitoring capability in the system has. For example, the monitoring capability (Generic Eavesdropper, book_flight, Response Time) has a set of available monitoring levels ML: \{0, 0.5, 1\}, representing sampling rates of 0%, 50%, and 100%, respectively.

\(ms \in MS\) and \(ms = ((m, s, q) \mapsto ml)\)

\(ms\) is a Monitoring Setting, which represents a selection of a monitoring for a monitoring capability. For example, \(((\text{Generic Eavesdropper, Response Time, Book Flight}) \mapsto 1)\) represents the Generic Eavesdropper monitor observing Response Time of the Book Flight service at monitoring level 1 (which in this case represents a 100% sampling rate). The set of all possible monitoring settings is denoted as \(MS\).

\(mc \in MC\) and \(mc = \{(m, s, q) \mapsto ml : ML \upharpoonright \forall (m, s, q) \in MCAP\}\)

\(mc\) is a Monitoring Configuration, which is a set of monitoring settings obtained by assigning each of the monitoring capabilities in a system with a monitoring level of its own. \(MCAP\) is the set of all available monitoring capabilities in the system. For example, one iTravel monitoring configuration may be: (book_flight Probe, book_flight Response Time) \(\mapsto 0.0\); (book_flight Probe, book_flight, Privacy) \(\mapsto 1.0\); (book_flight Probe, book_flight, Security) \(\mapsto 1.0\); (flight_info Probe, flight_info, Response Time) \(\mapsto 0.0\); etc...

\((s, q) \mapsto il|il \in \mathbb{R}\)

We use impact level \(il\) to denote the monitoring impact on a quality \(q\) of a service \(s\), where the impact level \(il\) is a percentage of the quality impact over the original quality (without the impact). For example, \(((s_1, q_1) \mapsto 0.5)\) means the level of \(q_1\) of \(s_1\) will be half of what it was - if \(q_1\) is response time, then response time would be doubled.
\[ msi = \{(s, q) \mapsto il : IL | \forall (s, q) \in SQ\} \]

\[ MI = \{ms \mapsto msi : \mathcal{P}(MIS)|\forall ms \in MS\} \]

\( msi \) represents the impacts that a monitoring setting \( ms \) has on each quality \( q \) of each service \( s \) in the system. For example, IT records may show that when monitor \( m_1 \) is configured to measure \( q_1 \) (response time) of service \( s_1 \) at monitoring level 0.5, it reduces \( q_1 \) of services \( s_1 \) and \( s_2 \) by 10%. The impacts of this monitor setting are \( \{(s_1, q_1) \mapsto 0.1, (s_2, q_1) \mapsto 0.1\} \). The set of impacts of all possible monitor settings in \( MS \) is \( MI \). For example, the above particular monitor setting’s impacts are a member of the overall \( MI \): \( \{((m_1, s_1, q_1) \mapsto 0.5) \mapsto \{(s_1, q_1) \mapsto 0.1, (s_2, q_1) \mapsto 0.1\}\} \in MI \).

\[ mo = \{(s, q) \mapsto il : IL | \forall (s, q) \in SQ\} \]

\[ MIS = SQ \rightarrow IL \]

\[ MO = \{m \mapsto mo : \mathcal{P}(MIS)|\forall m \in M\} \]

The monitoring overhead impact \( mo \) of a monitor \( m \) on a relevant service \( s \) occurs whenever the monitor is in use, regardless of the monitoring level of the monitor, or what qualities or services it is monitoring. Let \( SQ \) be all the service-quality pairs of concern in the system, the overheads a monitor has on every relevant quality \( q \) of every service \( s \) are \( mo = \{(s, q) \mapsto il : IL | \forall (s, q) \in SQ\} \).

For example, an interceptor \( m_1 \) that redirects messages of services \( s_1 \) and \( s_2 \) for analysis may reduce response time of all messages passing through it by 10%, whether or not those messages are actually analysed. Let \( MIS = SQ \rightarrow IL \), the set of all monitor overheads is the collection of all the monitors’ overheads, \( MO = \{m \mapsto \mathcal{P}(MIS)|\forall m \in M\} \). The overheads in the above example are a member of \( MO \): \( \{m_1 \mapsto \{(s_1, q_1) \mapsto 0.1, (s_2, q_1) \mapsto 0.1\}\} \in MO \), where \( q_1 \) represents response time.

\[ I_{s,q}(mc) = \sum_{ms \in mc} MI(ms)(s, q) \]

\[ O_{s,q}(mc) = \sum_{m \in M} MO(m)(s, q) | \exists mc_i \in mc, mc_i.m = m, mc_i.ml > 0. \]

\[ IM(mc)(s, q) = I_{s,q}(mc) + O_{s,q}(mc) \]

\[ IM(mc) = \{(s, q) \mapsto IM(mc)(s, q) | \forall (s, q) \in SQ\} \].
\[ MCI = \{ mc \mapsto IM(mc) \mid \forall mc \in MC \} \].

\[ I_{s,q}(mc) \] represents the set of monitoring instance impacts \( IM(ms) \) from a monitoring configuration \( mc \) for a given service \( s \) and quality type \( q \), and \( O_{s,q}(mc) \) represents the set of monitoring overhead impacts from a monitoring configuration \( mc \) on a given service \( s \) and quality type \( q \). The set of total impact on a given service \( s \) and quality type \( q \) for a monitoring configuration is \( IM(mc) \), the sum of the monitoring configuration’s monitoring instance impacts and overhead impacts. The set of impacts on all services and quality types is \( IM(mc) = \{(s, q) \mapsto IM(mc)(s, q) \mid \forall(s, q) \in SQ\} \). The set \( MCI \) represents the set of all possible monitoring configurations \( MC \) with their total impacts \( IM \).

\[ r = (s, q, ql, ml) \]

\( r \) represents a requirement that a quality \( q \) of a service \( s \) should be monitored with monitoring level \( ml \) whilst achieving a quality level \( ql \). For example, one requirement may be \( r = (s_1, q_1, 0.5, 0.5) \), representing the requirement to monitor quality type \( q_1 \) of service \( s_1 \) at a monitoring level \( ml \) of 0.5 or higher, whilst maintaining a quality level \( ql \) of 0.5 or higher. The quality level is modelled as a percent of the ideal, unimpacted quality level of that service.

\[ ri(s, q) \]

\( ri(s, q) \) gives a recorded impact on a quality \( q \) of a service \( s \) in a monitored system. The recorded impact is the difference between an ideal quality level \( IQL \) and a measured quality level \( MQL \) for a given Quality of Service (QoS). For example, if the ideal quality level for \textbf{response time} of \textit{book_flight} is 100% (as is usually, but not always the case), and the measured quality level for \textbf{response time} of \textit{book_flight} is 75%, then the recorded impact \( ri(book\_flight, response\_time) \) is 25%.

\[ rp = (s, q, ql, ml) \mapsto penalty, rp \in RP \]

\[ RP = \{(s, q, ql, ml) \mapsto penalty : \mathbb{R} \mid \forall(s, q, ql, ml) \in R\} \]
rp represents a requirement with a penalty. The penalty represents the cost to the service provider if that requirement is not met. RP is the set of all requirements with penalties in the system. For example, rp = (book_flight, Response Time, 0.9,1.0) \rightarrow 20 means that $20 must be paid if the response time of the book_flight service is less than 90% of the pre-computed ideal, or the sampling rate of monitoring is lower than 100%.

\[
util(s, q, ql, ml) = (\text{curmax } RP(s, q, ql, ml) - \sum_{lr \in LR(s, q, ql, ml)} (\text{curmax } - util(lr)),
\]

\[
LR(s, q, ql, ml) = \{(s, q, ql_1, ml_1) | \forall (s, q, ql_1, ml_1) \in R((ql_1 < q_1) \land (ml_1 \leq ml)) \lor ((ql_1 \leq q_1) \land (ml_1 < ml))\}
\]

\[
U = \{(s, q, ql, ml) \mapsto util(s, q, ql, ml)|(s, q, ql, ml) \in R\}
\]

util is a utility function which aggregates all requirements with penalties RP for a given service, qualitytype, qualitylevel and monitoringlevel. curmax represents the maximum available utility for a util, whilst LR is the set of requirements that will not be met due to either a quality level being too low (lr.ql < ql or a monitoring level being too low (lr.ml < ml). U is the utility function covering all requirements in R.

\[
R_{(s,q)} = \{(s_1, q_1, ql, ml)|(s_1, q_1, ql, ml) \in R \land s_1 = s \land q_1 = q\}
\]

R_{(s,q)} represents the set of all requirements on a particular service s and quality type q.

\[
IQL = \{(s, q) \mapsto iql, \forall s \in S, \forall q \in Q\}
\]

iql is an Ideal Quality Level, which is the best achievable quality level for a (service,quality) pair. For example (book_flight, response_time) \mapsto 5, means that the lowest possible response time for book_flight is 5 seconds.

\[
FQL_{mcon} = \{(s, q) \mapsto fql| fql = IQL(s, q) - MCI(mcon)(s, q)\}
\]

fql is a Final Quality Level for a monitoring configuration mcon - i.e. a quality of service that will be achieved when the monitoring described by mcon is performed.
\[ u(mc)(s, q) = \sum_{(s, q, ml) \in R_{(s, q)}} (\text{util}(m, s, q) \times a \times b), \text{ where} \]

\[ a = 1 \text{ if } FQL(s, q) \geq ql, \text{ or } a = 0 \text{ otherwise}; \]

\[ b = 1 \text{ if } (\exists m \in M, ms(m, s, q) \geq ml), \text{ or } b = 0 \text{ otherwise}. \]

\( u \) gives the utility provided by a particular service \( s \) and quality type \( q \) under monitoring configuration \( mc \). The components \( a \) and \( b \) represent requirements on the delivered final quality level and the monitoring level, respectively.

\[ u(mc) = \sum_{(s, q) \in SQ_{mc}} u(mc)(s, q), \text{ where} \]

\[ SQ_{mc} = \{(s, q) | \forall (m, s, q, ml) \in mc\} \]

\( u(mc) \) gives the total utility for the monitoring configuration \( mc \) as the sum of the utilities for all \( (s, q) \) pairs under that configuration.

\[ MCU = \{mc \mapsto u(mc) | mc \in MC\} \]

\( MCU \) is the set of all monitoring configurations with utility. That is, \( MCU \) maps each monitoring configuration \( mc \in MC \) to the net utility \( u(mc) \) of using that configuration.

\[ \text{optimal}_{mc} \in \{mc_1 | \forall mc_2 \in MC, MCU(mc_1) \geq MCU(mc_2)\} \]

\( \text{optimal}_{mc} \) is the optimal monitoring configuration (the monitoring configuration with the highest utility).

\[ DQ = \{(s, q) \mapsto mql | \forall s \in S, q \in Q\} \]

The delivered QoS \( DQ \) gives all monitored quality levels \( mql \) for each quality \( q \in Q \) of each service \( s \in S \) under a given monitoring configuration.

\[ RQ = \{mc \mapsto DQ\} \]

The multiset \( RQ \) gives a set of raw delivered QoS records \( mc \mapsto DQ \) from monitors in the system under a particular monitoring configuration \( mc \). \( RQ \) is essentially a set of monitoring records with each member of the set consisting of a monitoring configuration for that record along with that record’s set of measured qualities of service.
\( PQ = mc \mapsto AI \), where

\[
AI(s, q) = IQL(s, q) - AQ(s, q),
\]

and

\[
AQ(s, q) \mapsto aql|∀s ∈ S, q \in \mathcal{Q}, \text{ and}
\]

\[
aql = \text{average}(RQ(mc)(s, q))
\]

\( PQ \) gives the set of processed QoS records for a system, defined as a set of monitoring configurations \( mc \) and their resulting average impacts \( AI \). Each member in the set of average impacts \( AI(s, q) \) is defined as the ideal quality level \( IQL(s, q) \) minus the average quality level \( AQ(s, q) \). The average quality level \( AQ(s, q) \) is defined as the average of every raw quality record for the given monitoring configuration \( mc \), service \( s \) and quality type \( q \), \( aql = \text{average}(RQ(mc(s, q))) \).

\( MQL(s, q) \)

\( MQL(s, q) \) gives a measured quality level for a particular service \( s \) and quality type \( q \). This measured quality level is the quality level as monitored by one monitor for a single service invocation in the system, and is returned as a percentage of the ideal quality level \( IQL \) for a particular QoS. For example, if the ideal quality level for response time of book_flight is 10 seconds, and a book_flight execution is measured to be 11 seconds, then the measured quality level \( MQL(\text{book_flight}, \text{response time}) \) would be 10%.

\( ri(s, q) \)

\( ri(s, q) \) gives a recorded impact on a quality \( q \) of a service \( s \) in a monitored system. The recorded impact is the difference between an ideal quality level \( IQL \) and a measured quality level \( MQL \) for a given QoS. For example, if the ideal quality level for response time of book_flight is 100% (as is usually, but not always the case), and the measured quality level for response time of book_flight is 75%, then the recorded impact \( ri(\text{book_flight}, \text{response time}) \) is 25%.
Appendix C

Survey of Web Service Monitoring Literature

This section provides a summary of our WS monitoring literature survey.

Research and development efforts on WS monitoring can be classified by:

- The monitoring technique(s) used [Technique];
- The location of the monitor(s) (e.g. intermediary message interception, service-side monitoring process...) [Monitor location];
- The abstraction level of the approach (e.g. generic technique, implemented framework) [Abstraction Level];
- Whether the monitoring is Dynamic, where the monitoring system allows for runtime reconfigurations either manually [M] or automatically [A], based on results it receives [Dyn.];
- Whether the approach allows for balancing monitoring impacts with monitoring coverage either for complete [C] or incomplete [I] coverage [Imp. Trade-off];
- Who the beneficiary of the monitoring is [Monitoring for];
- Who performs the monitoring [Monitoring by];
- What is monitored [Monitoring of];
- Whether the approach models monitoring impacts according to business or SLA costs [Bus-Cost];

- Whether the system unnecessarily uses an interceptor-type monitor [Bad Proxy]; and

- The impact of the approach [Imp.].

Table C.1 presents a comparative summary of all the literature reviewed in this chapter. The rows with a white background contain research efforts that present solutions for monitoring either individual services or an Service Oriented Architecture (SOA) process, be it a Web Services Business Process Execution Language (WS-BPEL) process, a generic SOA process, or the conversations between services. The rows with a light grey background contain research efforts that present solutions for monitoring SLAs or service consumer requirements. The rows with a black background contain research efforts that present monitoring of service provider system resources.

For each technique, we have listed the types of monitors used according to our taxonomy, reported in the **Technique** column. We have identified those systems that used proxy-based monitoring when eavesdropper-based monitoring would have sufficed, reported in the **Bad Proxy** column. We have also identified those works that mention the performance impact of their monitoring solution (Imp.).

Detailed reviews of each of the works in Table C.1 are provided in Chapter 3. Below, we present a summary of each of the properties (corresponding to a column in Table C.1) we consider important, across all of the works surveyed.

The range of monitoring techniques used by the works in our survey was widely varied, with ten works using Eavesdropper-based monitors, nine works using Interceptor-based monitors, six works using Probe-based monitors, and ten works using all three classes of monitor. This shows that there is a good distribution of monitor classes in use. Whilst the most common individual monitoring class in use was the Eavesdropper, there was still a significant amount of Probe-based and Interceptor-based monitoring used. Our experiments in Chapter 4 show that the Probe and Interceptor classes in particular have significant monitoring impacts. Therefore, our optimisation approach may be applicable
in many of these cases.

26 of the works surveyed used the service provider as the **monitor location**. Eight placed the monitor at an intermediary, which was typically a proxy between the provider and consumer. Six works placed the monitor on the consumer-side, and one was based at a dedicated third party. These results show that the majority of monitoring was being performed on the web service provider side, which aligns with the design of our optimisation technique, since hosting monitors at the provider is likely to have an impact on the services offered by that provider.

26 of the works surveyed was at the **abstraction level** of a framework. 11 of these were implemented or partially implemented, 10 were prototyped and five were abstract. Of the remaining works, six were generic techniques, two were generic architectures [12, 38] and two were implemented architectures [34, 20], and one was a generic methodology [15]. This shows that most of the research effort reflected in our survey is towards developing new or improved monitoring frameworks, and that most of these frameworks have been implemented or at least prototyped.

Only 11 of the works surveyed provided **dynamic** monitoring reconfiguration abilities. Of these, eight provided manual reconfiguration abilities and three were automated. This shows that most of the works do not consider modification of the monitoring system at run-time based on monitoring results.

Only eight of the works surveyed allowed for **impact trade-off**. Four of these provided the ability to perform impact trade-off for complete monitoring coverage. That is, these works provided a mechanism to trade-off between different monitors to minimise monitoring cost, but did not consider situations in which it is more beneficial to not perform monitoring. The remaining four works did provide impact trade-off with incomplete monitoring coverage.

24 of the works were designed to be **monitored for** the benefit of the service provider. 16 monitored for the benefit of the service consumer, whilst six monitored for the benefit of the BPEL owner (who could be either the service provider, service consumer or a third party). This shows that the set of beneficiaries for monitoring is widely distributed between service consumers and service providers.

24 of the works were designed to be **monitored by** the service provider, with 10
designed to be monitored by the service consumer, eight the BPEL owner, four a third party and three an unspecified intermediary. Since our approach is targeted at optimising the service provider’s suite of monitors, it may be applicable to most of the works here (the 24 provider and 8 BPEL-owner cases).

13 of the works were designed for monitoring of services, a further 13 were designed for monitoring of service compositions or processes, eight were designed to monitor SLAs, seven were designed to monitor the local resources of the service provider (such as hardware or network utilisation), four were designed to monitor service interactions (such as service conversations) and one was designed to monitor consumer data [22]. These results show that there is a wide range of targets for web service monitoring, which would require different monitoring techniques or classes of monitor to be used, and are hence likely to benefit from our optimisation approach.

Only two of the works surveyed considered the business cost of monitoring (that is, the cost to the business due to monitoring impacts on SLAs or other business requirements with financial incentives) [31, 68]. As such, our approach may be useful even if just to provide a model or estimate of these costs.

Of the 23 works that used proxy (interceptor)-based monitors, four were considered to have bad proxies. That is, these works used blocking interceptor-based monitors where probes or eavesdroppers would have sufficed. This is an indication that some systems could be improved by using our monitoring classification and impact profiles to select appropriate monitor classes that minimise monitoring impacts, whilst maintaining the same capabilities.

Only five of the 39 works provided a quantified impact: “<5%” due to using an existing proxy [6], “1-5%” due to using eavesdroppers [12], “up to 40%” due to probing [61], “1–7%” due to eavesdropping [46], and “10-100%” due to performing monitoring analysis on WS-BPEL servers [9]. Three works stated the impact as “negligible” due to use of eavesdropping or existing proxies. The remaining works did not report on the performance impact of their monitoring solutions. These results show that, in general, designers of distributed monitoring systems have placed little importance on performance impacts.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Abdu et al., 2002</td>
<td>All</td>
<td>Provider</td>
<td>Technique, implemented</td>
<td>M</td>
<td>C</td>
<td>Provider</td>
<td>Provider</td>
<td>Services</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Ameller and Franch, 2008</td>
<td>Probe</td>
<td>Provider</td>
<td>Framework, prototyped</td>
<td>No</td>
<td>No</td>
<td>Provider</td>
<td>BPEL owner</td>
<td>Services</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Bai et al., 2009</td>
<td>All</td>
<td>SOAP engine, provider</td>
<td>Framework, prototyped</td>
<td>M</td>
<td>I</td>
<td>Provider</td>
<td>Provider</td>
<td>Compositions, Services</td>
<td>No</td>
<td>No</td>
<td>&lt;5%</td>
</tr>
<tr>
<td>Barbon et al., 2006</td>
<td>Interceptor</td>
<td>BPEL server</td>
<td>Framework, implemented</td>
<td>No</td>
<td>No</td>
<td>BPEL owner</td>
<td>BPEL owner</td>
<td>BPEL Process</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>Baresi and Guinea, 2005</td>
<td>Interceptor</td>
<td>Custom BPEL Intermediary</td>
<td>Framework, partially implemented</td>
<td>A</td>
<td>I</td>
<td>BPEL owner</td>
<td>BPEL owner</td>
<td>SOA Process</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>Baresi and Guinea, 2011</td>
<td>Eavesdropper</td>
<td>Provider</td>
<td>Framework, prototyped</td>
<td>M</td>
<td>I</td>
<td>Provider</td>
<td>Provider</td>
<td>BPEL Process</td>
<td>No</td>
<td>No</td>
<td>10–100%</td>
</tr>
<tr>
<td>Benbernou et al., 2007</td>
<td>Probe</td>
<td>Intermediary, not specified</td>
<td>Framework, abstract</td>
<td>No</td>
<td>No</td>
<td>Consumer</td>
<td>Unspecified intermediary</td>
<td>Consumer’s private data</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Benharref et al., 2009</td>
<td>Eavesdropper</td>
<td>Mobile</td>
<td>Architecture</td>
<td>No</td>
<td>No</td>
<td>Provider</td>
<td>Provider</td>
<td>Compositions, Services</td>
<td>No</td>
<td>No</td>
<td>1-5%</td>
</tr>
<tr>
<td>Bertolino et al., 2007</td>
<td>All</td>
<td>Provider</td>
<td>Technique, generic</td>
<td>A</td>
<td>Yes(^1)</td>
<td>Provider</td>
<td>Provider</td>
<td>SLAs</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Brenner et al., 2007</td>
<td>Probe, Eavesdropper</td>
<td>Consumer</td>
<td>Methodology</td>
<td>No</td>
<td>No</td>
<td>Consumer</td>
<td>Consumer</td>
<td>Service Correctness</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Comuzzi et al., 2009</td>
<td>All</td>
<td>Provider</td>
<td>Framework, prototyped</td>
<td>No</td>
<td>No</td>
<td>Consumer</td>
<td>Provider</td>
<td>SLAs</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Comuzzi and Spanoudakis, 2010</td>
<td>All</td>
<td>Consumer, Provider</td>
<td>Framework, prototyped</td>
<td>M</td>
<td>C</td>
<td>Consumer, Provider</td>
<td>Provider, Consumer</td>
<td>SLAs</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>da Cruz et al., 2004</td>
<td>Interceptor</td>
<td>SOAP Intermediary</td>
<td>Architecture, partially implemented</td>
<td>No</td>
<td>No</td>
<td>Consumer, Provider</td>
<td>Third Party</td>
<td>Atomic Service</td>
<td>No</td>
<td>Yes</td>
<td>Negl.</td>
</tr>
<tr>
<td>Ezenwoye and Sadjadi, 2006</td>
<td>Interceptor</td>
<td>BPEL server</td>
<td>Framework, prototyped</td>
<td>No</td>
<td>No</td>
<td>Consumer</td>
<td>BPEL process server</td>
<td>WS Process</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\(^1\)Inverted

Continued on Next Page...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Eavesdropper</td>
<td>Provider, Consumer</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>Provider, third parties</td>
<td>Keller and Ludwig, 2002</td>
<td></td>
</tr>
<tr>
<td>Interceptor</td>
<td>Provider</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>Consumer, Intermediary</td>
<td>Lazo et al., 2004</td>
<td></td>
</tr>
<tr>
<td>Custom Intermediary</td>
<td>Provider</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>Provider</td>
<td>Lazo et al., 2006</td>
<td></td>
</tr>
<tr>
<td>Framework, prototyped</td>
<td>BPEL owner</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
<td>No</td>
<td>Consumer, Provider</td>
<td>Li et al., 2006</td>
<td></td>
</tr>
<tr>
<td>Framework, partially implemented</td>
<td>BPEL owner</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>Provider</td>
<td>Ludwig et al., 2004</td>
<td></td>
</tr>
<tr>
<td>Framework, implemented</td>
<td>BPEL owner</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>Provider</td>
<td>Li et al., 2006</td>
<td></td>
</tr>
<tr>
<td>Intermediary on Provider side</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>Provider</td>
<td>Eavesdropper</td>
<td></td>
</tr>
<tr>
<td>Implementer</td>
<td>Custom Intermediary</td>
<td>M</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Consumer, Provider</td>
<td>Lazo et al., 2006</td>
<td></td>
</tr>
<tr>
<td>Provider, Designer, Provider</td>
<td>Provider</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>Provider</td>
<td>Leitner et al., 2010</td>
<td></td>
</tr>
<tr>
<td>Framework, prototyped</td>
<td>SOAP Intermediary</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th><strong>Source</strong></th>
<th><strong>Technique</strong></th>
<th><strong>Monitor location</strong></th>
<th><strong>Abstraction Level</strong></th>
<th><strong>Dyn.</strong></th>
<th><strong>Imp. Trade-off</strong></th>
<th><strong>Monitoring for</strong></th>
<th><strong>Monitoring by</strong></th>
<th><strong>Monitoring of</strong></th>
<th><strong>Bus-Cost</strong></th>
<th><strong>Bad Proxy</strong></th>
<th><strong>Imp.</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mahbub and Spanoudakis, 2004, Mahbub and Spanoudakis, 2005</td>
<td>Eavesdropper</td>
<td>Custom BPEL Intermediary</td>
<td>Framework, prototyped</td>
<td>No</td>
<td>No</td>
<td>BPEL owner</td>
<td>BPEL owner</td>
<td>SOA Process</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>Molina-Jimenez et al., 2004</td>
<td>Probe</td>
<td>Third party</td>
<td>Architecture, abstract</td>
<td>M</td>
<td>No$^2$</td>
<td>Consumer, Provider</td>
<td>Third Party</td>
<td>SLAs</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Narendra et al., 2008</td>
<td>Interceptor</td>
<td>Provider</td>
<td>Technique, prototyped</td>
<td>M</td>
<td>I</td>
<td>Provider</td>
<td>Provider</td>
<td>Business Processes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>Netto et al., 2007</td>
<td>All</td>
<td>Provider</td>
<td>Framework, implemented</td>
<td>No</td>
<td>No</td>
<td>Provider</td>
<td>Provider</td>
<td>Resource</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>Panahi et al., 2008</td>
<td>All</td>
<td>Provider</td>
<td>Framework, partially implemented</td>
<td>M</td>
<td>C</td>
<td>Consumer, Provider</td>
<td>BPEL owner</td>
<td>Provider Resources, Services</td>
<td>No</td>
<td>N/A</td>
<td>1-7%</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Qiang et al., 2006</td>
<td>Eavesdropper</td>
<td>Consumer, Intermediary, Provider</td>
<td>Framework, abstract</td>
<td>No</td>
<td>No</td>
<td>Consumer, Provider</td>
<td>Consumer, Provider</td>
<td>Services</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>Raimondi et al., 2007</td>
<td>Interceptor</td>
<td>Consumer, Provider</td>
<td>Framework, implemented</td>
<td>No</td>
<td>No</td>
<td>Consumer</td>
<td>Consumer, Provider</td>
<td>Service conversations</td>
<td>No</td>
<td>No</td>
<td>Negl.</td>
</tr>
<tr>
<td>Ranganathan and Dan, 2005</td>
<td>Probe</td>
<td>Provider</td>
<td>Framework, implemented</td>
<td>No</td>
<td>No</td>
<td>Provider</td>
<td>Provider resources</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>Robinson, 2003</td>
<td>Eavesdropper</td>
<td>Unspecified</td>
<td>Framework, abstract</td>
<td>No</td>
<td>No</td>
<td>User requirements</td>
<td>Provider, consumer, third party</td>
<td>WS messages, services</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Rouached and Godart, 2006</td>
<td>Eavesdropper, Interceptor</td>
<td>SOAP Intermediary</td>
<td>Framework, implemented</td>
<td>No</td>
<td>No</td>
<td>BPEL owner</td>
<td>Provider, BPEL owner</td>
<td>WS Process</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>Sahai et al., 2002</td>
<td>SOAP Interceptor</td>
<td>Consumer, Provider</td>
<td>Framework, prototyped</td>
<td>No</td>
<td>No</td>
<td>Consumer, Provider</td>
<td>Consumer, Provider</td>
<td>SLAs</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>Shao et al., 2010</td>
<td>All</td>
<td>Provider</td>
<td>Framework, implemented</td>
<td>A</td>
<td>No³</td>
<td>Cloud provider</td>
<td>Cloud provider, Consumer</td>
<td>Applications, Resources</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

³Possible future work
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Skene et al., 2007</td>
<td>Eavesdropper</td>
<td>Consumer, Provider</td>
<td>Technique, abstract</td>
<td>No</td>
<td>No</td>
<td>Provider, Consumer</td>
<td>Provider, Consumer</td>
<td>Service interactions</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>Singh et al., 2006</td>
<td>Probe, Eavesdropper</td>
<td>Provider</td>
<td>System, implemented</td>
<td>No</td>
<td>No</td>
<td>Provider</td>
<td>Provider</td>
<td>Distributed System</td>
<td>No</td>
<td>N/A</td>
<td>40%</td>
</tr>
<tr>
<td>Truong et al., 2006</td>
<td>All</td>
<td>Provider</td>
<td>Framework, prototyped</td>
<td>No</td>
<td>No</td>
<td>Provider</td>
<td>Provider, Consumer</td>
<td>Services, Infrastructure</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Zeng et al., 2007</td>
<td>N/A</td>
<td>Provider</td>
<td>Framework, implemented</td>
<td>No</td>
<td>No</td>
<td>Provider</td>
<td>Provider</td>
<td>SOA Process</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Zhang et al., 2008</td>
<td>N/A</td>
<td>Provider</td>
<td>Technique, implemented</td>
<td>No</td>
<td>C</td>
<td>Provider</td>
<td>Provider</td>
<td>Services</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

4Up to 40% extra CPU usage. Superlinear CPU usage with probing
Appendix D

Introduction to Genetic Algorithms

A Genetic Algorithm is a class of heuristic search that attempts to find an optimal solution through a simulation of the evolutionary process [25, 53]. A Genetic Algorithm (GA) works by creating a population (set) of individuals (candidate solutions), and then continuously measuring those individuals’ fitness (quality, in our case total utility) and selecting the fittest (highest utility) individuals for mutation (small changes, e.g. taking a small step in one dimension in an n-dimensional space) and crossover (reproduction). During crossover, genes (parameters) of two parent individuals are taken and combined to create a child in the next generation. The GA continues this process of selection, mutation, and crossover until some stopping criterion is reached. A stopping criterion may be time or generations elapsed, detection of a plateau, or some acceptable fitness level reached. The fitness of each member of the population is taken into consideration when determining which should be crossed over or mutated, which should be passed to the next generation without change, and which should be discarded. This means that there is a higher probability of two members of the population which yield a high fitness reproducing, and a higher probability of some member of the population with a low fitness being discarded.

Each parameter in an optimisation problem becomes a gene in the GA, and the allowable values for that parameter are the allowable values for that gene. For example, if we have a hill-climber in three dimensions, there will be three genes (parameters) for each
individual in the GA. The total set of genes is represented as a chromosome (a string of
genes). For the hill-climber example, each gene would be able to hold any value that is
valid (i.e. exists on the 3D plane) for the search space.

A GA requires a fitness function that evaluates an individual and returns a value that
represents how fit (good) that individual is. For the hill-climber example above, this might
be the value of the Z-dimension for the individual.

As mentioned, a GA randomly creates a set of individuals (a population), and mea-
sures the fitness of each individual in the population. This set of individuals is the first
generation. The fittest individuals of this first generation are selected for crossover (re-
production). This means that, for two individuals, their chromosome will be spliced, and
one part of each individual’s chromosome will be combined to produce offspring for the
next generation. For example, if two individuals selected for crossover are (0,0,0) and
(1,1,2), then one child could be (0,1,2), and another could be (1,1,0). The intention of
this operation is to successively select the genes that yield the highest fitness. These
functions differ in how many points and where they splice the chromosome[53]. Single
Point methods select a point in the chromosome, divide it into two, and swap each side
of the two parents (e.g., (0,0) and (1,1) may yield (0,1) and (1,0)). Two Point methods
select two points in the chromosome, divide it into three, and swap the centre section of
each chromosome to create children (e.g., (0,0,0) and (1,1,1) will yield (0,1,0) and (1,0,1)).
Uniform methods randomly select individual genes from each parent to create a child
(e.g., (0,0,0) and (1,1,1) may yield (0,1,1) or any other combination of 0’s and 1’s). For
Roulette-wheel based methods, the probability of an individual being selected as a parent
is proportional to the fitness of that individual - i.e., the fittest individuals are more likely
to be selected as parents. For Tournament-based methods, the fittest of a set of individuals
is selected as a parent - i.e. randomly take a subset of individuals and select the fittest.
For Random-based methods, individuals are randomly selected as parents.

In addition to crossover, individuals are randomly mutated, so that (0,0,0) might
become (0,1,0) for the next generation. Once all crossovers and mutations have been
performed, a new generation exists. The fitness of each individual in this population is
evaluated, and another round of crossover and mutation is performed. This continues until
some pre-specified termination criteria is met. In summary, the steps performed are:
(1) Initialisation, in which the GA object is created and a population is randomly generated

(2) Evaluation, in which each member of the population’s fitness is measured (i.e. the total utility of that member is calculated)

(3) Crossover, in which parents are selected from the population, and they are crossed over (i.e. parts from each parent are taken and combined together) to produce a child

(4) Mutation, in which members of the population are randomly modified

(5) Evaluation, and if a stopping criteria is not met, go back to step 3.

The fitness of each member of the population is taken into consideration when determining which should be crossed over (step 3) or mutated (step 4), which should be passed to the next generation without change, and which should be discarded. This means that there is a higher probability of two members of the population which yield a high fitness reproducing, and a higher probability of some member of the population with a low fitness being discarded.
Publications


(2) Garth Heward, Ingo Muller, Jun Han, Jean-Guy Schneider and Steven Versteeg, "Optimizing the Configuration of Web Service Monitors", International Conference on Service-Oriented Computing (ICSOC), pp 587-595, San Francisco, United States, 2010.

(3) Garth Heward, Ingo Muller, Jun Han, Jean-Guy Schneider and Steven Versteeg, "A Case Study on Optimizing Web Service Monitoring Configurations", Workshop on Engineering Service-Oriented Applications (WESOA), pp 4-14, San Francisco, United States, 2010.

(4) Garth Heward, Jun Han, Jean-Guy Schneider and Steven Versteeg, "Run-time management and optimization of web service monitoring systems", International Conference on Service-Oriented Computing and Applications (SOCA), pp 1-6, Irvine, United States, 2011.