QoS Enforcement for Web Services in Dynamic Networks

Nir Shemy
QoS Enforcement for Web Services in Dynamic Networks

Research Thesis

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science

Nir Shemy

Submitted to the Senate of the Technion — Israel Institute of Technology
ADAR, 5769 Haifa March, 2009
This Research Thesis was done under the supervision of Assoc. Prof. Roy Friedman in the Department of Computer Science

I wish to express my deep gratitude to Roy Friedman for having the privilege to work under his supervision throughout the course of my research. It is his guidance, experience, help and support that made this research possible.

I would like to thank my parents, Sarah and Menachem Shemy, for their endless support and encouragement throughout the period of my research.

The generous financial help of the Technion is gratefully acknowledged
# Contents

Abstract ................................. 1

Notation and Abbreviations .......... 3

1 Introduction ........................... 5

2 The Architecture of SLAMES ...... 9
   2.1 Overview .......................... 9
   2.2 The Proxy-based Approach ...... 11
   2.3 High-Level Architecture ........ 12
   2.4 Use Cases ....................... 13

3 SLAMES Policy Language .......... 15
   3.1 Policy Conditions ................ 16
   3.2 Policy Actions ................... 18
   3.3 Policy Configuration .......... 20

4 SLAMES Implementation .......... 23
   4.1 The Engine ....................... 23
   4.2 The Management API .............. 24
   4.3 Persistency Layer ............... 25
   4.4 Scalability Issues ............... 26
   4.5 Engine Extensions ................ 27
       4.5.1 Adding a new action ........ 27
       4.5.2 Adding monitored variables .. 27
       4.5.3 Adding event-based conditions. 28
   4.6 Future Engine Optimizations .... 28
   4.7 Class Diagram ................... 29
## 5 Performance Evaluation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Evaluation Goals</td>
<td>31</td>
</tr>
<tr>
<td>5.2 Evaluation Environment</td>
<td>31</td>
</tr>
<tr>
<td>5.2.1 Machine Setup</td>
<td>31</td>
</tr>
<tr>
<td>5.2.2 Evaluated Policies</td>
<td>32</td>
</tr>
<tr>
<td>5.2.3 Evaluated Web Services</td>
<td>32</td>
</tr>
<tr>
<td>5.3 Performance Evaluation Results</td>
<td>32</td>
</tr>
<tr>
<td>5.3.1 Performance of Different Policies</td>
<td>32</td>
</tr>
<tr>
<td>5.3.2 The Performance Effect of Other Monitored Services</td>
<td>34</td>
</tr>
<tr>
<td>5.3.3 Local vs. Remote Installation</td>
<td>35</td>
</tr>
<tr>
<td>5.3.4 File vs. Database Based Persistency</td>
<td>36</td>
</tr>
<tr>
<td>5.3.5 Performance of Java Code vs. Jython Scripts</td>
<td>38</td>
</tr>
<tr>
<td>5.4 Evaluation Summary</td>
<td>38</td>
</tr>
</tbody>
</table>

## 6 Related Work

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 SLAs for Web Services</td>
<td>39</td>
</tr>
<tr>
<td>6.2 Network Monitoring</td>
<td>41</td>
</tr>
<tr>
<td>6.3 QoS Issues In Other Frameworks</td>
<td>42</td>
</tr>
</tbody>
</table>

## 7 Summary and Future Directions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Summary</td>
<td>43</td>
</tr>
<tr>
<td>7.2 Future Research Directions</td>
<td>44</td>
</tr>
</tbody>
</table>

## A Slames Policy Schema Definition

## B Evaluated Policy Definitions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1 Throughput-based Policy</td>
<td>51</td>
</tr>
<tr>
<td>B.2 Latency-based Policy</td>
<td>51</td>
</tr>
<tr>
<td>B.3 Intensive Policy</td>
<td>52</td>
</tr>
</tbody>
</table>

## References

## Hebrew Abstract
List of Figures

2.1 Typical service producer-consumer network topology .................... 9
2.2 SLAMES-based service producer-consumer network topologies ........ 10
2.3 Proxy-based Abstraction Layer for Web Services ....................... 11
2.4 SLAMES high-level architecture ...................................... 13

3.1 The structure of a SLAMES policy .................................... 16
3.2 A Valid WS-Policy Definition with SLAMES .......................... 17
3.3 SLAMES Event-based Condition Example ............................. 17
3.4 SLAMES Time-based Condition Example ............................. 18
3.5 Examples of SLAMES Actions ........................................ 20
3.6 Example of a Policy Configuration .................................... 20

4.1 The Primary Classes in SLAMES ...................................... 30

5.1 Different Policies Evaluation ........................................ 33
5.2 Monitored services effect on a single service performance .......... 35
5.3 Local vs. Remote Installation Response Times ....................... 36
5.4 File vs. Database Persistency Layer Performance ................... 37
List of Tables

3.1 SLAMES Monitored Variables ........................................ 19
3.2 SLAMES Control Variables ........................................... 21
3.3 SLAMES Configuration Parameters ................................. 21

5.1 Average Overhead of Evaluated Policies ......................... 34
5.2 Java vs. Jython Execution Time ................................. 38
Abstract

Internet growth during the last decades encouraged the development of new methods for using the web environment. Web services are one of these methods, providing a web-based API for remote execution of code over a network. They are very popular over the Internet and become an integral part of e-commerce business.

As web services gained more popularity, quality of service requirements for security, reliability and performance were crucial for the adoption of this technology in the business world. While some of them became a standard, there is no single acceptable standard for performance supported by the web services frameworks. Moreover, the enforcement of performance by existing frameworks is usually concerned with the service provider and rarely concerned with the consumer and the network environment.

We present SLAMES, a novel system providing performance provisioning and enforcement for web services. Unlike most existing frameworks, it ensures that the required performance is maintained in the consumer environment, including network delays. As such, it is capable of solving some network problems that cannot be addressed by existing frameworks installed on the side of the service provider. It also provides a new approach for ensuring quality of service by a third party rather than the service provider itself.

SLAMES provides a scalable solution for monitoring Web services using a proxy approach, making it as transparent as possible to both service provider and service consumer. It is compatible with existing standards and yet provides flexible configuration using scripting language capabilities.
## Notation and Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DB</td>
<td>Database</td>
</tr>
<tr>
<td>DSR</td>
<td>Dynamic Source Routing</td>
</tr>
<tr>
<td>FSB</td>
<td>Front Side Bus (used in computer hardware)</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>ISP</td>
<td>Internet Service Provider</td>
</tr>
<tr>
<td>JavaEE</td>
<td>Java Enterprise Edition</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>MANET</td>
<td>Mobile Ad-hoc Network</td>
</tr>
<tr>
<td>NPM</td>
<td>Network Performance Metrics</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality-of-Service</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SNMP</td>
<td>Simple Network Management Protocol</td>
</tr>
<tr>
<td>SOA</td>
<td>Service-Oriented Architecture</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Discovery, Description and Integration</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>WS</td>
<td>Web Service</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Description Language</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>MHz</td>
<td>a frequency rate of $10^6/\text{second}$</td>
</tr>
<tr>
<td>GHz</td>
<td>a frequency rate of $10^9/\text{second}$</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

Over the last two decades, communication networks have become an integral part of everyone’s life. Most people in modern countries are consumers of mobile network services using their cellular phones, connect to the Internet using their personal computer and even use sophisticated TV services using cable-based or satellite-based communication networks. For most organizations, online services play a crucial role in their work environment. Fundamental services, such as emails, phones and video conferences are widely used. A relatively new kind of business model, called e-business (electronic business) or e-Commerce (electronic commerce) is a major and ever-growing part of today’s business world.

In order to satisfy the needs of all these service consumers, frameworks were built to provide a quick creation and consumption of reliable cross-platform services. The industry is evolving into a new method for system development and integration called Service-Oriented Architecture (SOA) [4]. In the context of software development, SOA supporting frameworks provide a programming paradigm for building a set of reusable, loosely-coupled software elements called services.

The Web services framework [16] provides the means to create, publish, describe and invoke online services, and therefore widely used in SOA frameworks. It is built upon several standards in order to ensure that services could be used regardless of the operating system, programming language or any other middleware technology.

Web services use the Web Service Description Language (WSDL) standard [6] for describing the service interface, the Universal Discovery, Description and Integration (UDDI) [25] registry for advertising and querying for existing services, the HyperText Transfer Protocol (HTTP) [8] for communication and the Simple Object Access Protocol (SOAP) [32] as a serialization format over HTTP.

When a consumer application needs some service, it should retrieve the WSDL document describing it. The consumer can query for this document in some UDDI registry, or get it directly from the service provider if its location is known. The WSDL document contains information about the real location of the service provider, the protocols supported by the service (pure HTTP, SOAP, etc.) and also a description of the service interface - the various operations supported by the service, their input and output parameters and errors that may occur during the invocation of the operations. The information about the interface is commonly referred as the functional requirements of a service.
When it comes to real-world scenarios where companies provide and consume critical services for their business, these requirements are not enough. Consider a case when an organization cannot choose the best business decision because the information needed for having that decision was delayed or was not received at all. In order to be suitable for real-world scenarios, service providers should guarantee some level of quality from the service. These guarantees are called the non-functional requirements of a service, and often referred as Service Level Agreements (SLA). There are non-functional requirements from various aspects, such as security, reliability, availability, performance, etc.

There are additional standards defined in order to describe these non-functional requirements. These standards are optional extensions to the original Web service model, and most Web services framework vendors choose to implement only a subset of them. The most important standards are WS-Policy [35], which provides an XML-based schema for defining policies, and WS-PolicyAttachment [36], which provide a standard way to associate a policy with a service within its WSDL description document or as part of its declaration in the UDDI registry.

The WS-Policy does not define any non-functional requirements that a service provider is committed to deliver. It is only a skeleton schema for aggregating policy assertions, which are non-functional requirements defined in other standards. Security token definition (WS-Security), trust model (WS-Trust), reliable messaging (WS-ReliableMessaging), interoperability support (WS-I) and transaction support (WS-Transaction) are examples of popular standards covering some of the aspects addressed in non-functional requirements. However, there are no official standards for availability and performance quality of service issues. Web services framework vendors use their own propriety model for providing these types of service quality.

There have been several studies in the direction of performance SLAs, e.g., [22, 33, 11, 21, 20, 24]. Most of them concentrate on the problem of choosing the best service by comparing the policies of several service providers and finding the best service for a specific consumer. This problem is known as Service Binding, or Service Matching. However, they do not talk about the ability to continuously monitor the performance and to ensure that service consumers get the quality of service that service providers guaranteed to deliver according to their advertised policies.

Two major studies regarding continuous monitoring and enforcement of SLAs were carried out by Tosic et al., proposing the Web Service Offerings Language [30] and by A. Keller and H. Ludwig, proposing the WSLA language [19] and framework [15]. Both studies suggest flexible languages for description of policies, including performance aspects. They also provide a framework that monitors and enforces these constraints. However, their works are focused on the ability of the service provider to stand up with its quality of service obligations and do not concentrate on the quality of service from the side of the service consumer. The service consumer might not get the desired quality of service due to external factors that are not under the direct responsibility of the service provider. For instance, network congestions in the path between the provider and the consumer, or an unstable network connection might prevent the consumer from getting the quality of service it was promised. Such problems are more acute when working in dynamic networks, which suffer from frequent changes in the network topology, making the network connectivity and the network delays a major factor in the evaluation of service performance.
In this thesis we present SLAMES, a Service Level Agreement Monitoring & Enforcement System. SLAMES uses a new rich model for defining availability and performance assertions. It also introduces a new approach for building web service policy enforcement systems. We discuss later the pros and cons of the new approach compared to the classic approach currently used for implementing these systems.

SLAMES monitors the performance from the consumer environment, making sure that the service consumer gets its promised quality of service rather than the service provider providing it. With this approach, SLAMES is able to provide solutions to some of the problems that cannot be detected or handled on the service provider side. For instance, SLAMES can fix an unstable network path between the service consumer and the service producer in dynamic networks by re-invoking Dynamic Source Routing (DSR) path discovery in order to find a new path, probably more stable for the near future, between the parts. Other known techniques from the world of network monitoring are now feasible for use only because SLAMES runs from the side of the consumer.

In order to provide flexible capabilities and yet be compatible with the existing standards, SLAMES integrates an XML-based policy language with Jython scripting language. The combination of these extremely different languages enables writing sophisticated policies that have the power of a programming language. SLAMES policies can be used along with existing QoS standard policies such as WS-Security or WS-Trust under the same definition of a policy written according to the WS-Policy standard.

SLAMES implementation provides a fairly scalable solution to the monitoring problems while minimizing the changes needed to be done in both service provider and service consumer parties by using a proxy-based approach for intercepting the communication between them. This way, service providers and consumers are encouraged to use SLAMES, as it does not require them to make any considerable change to their existing code.

This thesis is organized as follows: in chapter 2, we present the high-level architecture of SLAMES and demonstrate how SLAMES can be used to detect and solve some typical network problems. In chapter 3, we describe in detail how to write a policy configuration for SLAMES. Chapter 4 deals with the implementation details of SLAMES, discusses about the decisions made when building the system, how to extend it and suggests several performance optimizations to the system. Chapter 5 provides some statistics about the overhead of using this system. In chapter 6, we compare the system to other related studies in this area. Chapter 7 provides some conclusions and offers some directions for future study in this area.
Chapter 2

The Architecture of SLAMES

2.1 Overview

When using the Web services model, there are usually two parties participating in the mechanism: the service provider and the service consumer. The service provider and the service consumer are connected through some kind of network, and use this connection in order to communicate between each other directly.

SLAMES is a system for provisioning and enforcement of service level agreements in Web services. It is destined to monitor performance and availability quality aspects, but can also be used for other aspects such as security, trust, reliable messaging, etc. SLAMES is designed as a middle-tier component between the service provider and the service consumer. It is aimed at monitoring the real performance of the service provider for a specific service consumer including network delays between them. Consequently, the system should be installed on the same host as the client or on a machine, which is very close to the consumer in the network topology, such as the same local network of the Web service consumer.

Unlike most existing frameworks for Web services performance monitoring, SLAMES does not need to be part of the service provider environment. It can be used by the service consumer in situations when the consumer has some control over the network path to the provider, or by a third party with network control capabilities (like an ISP, or a network infrastructure provider).

![Typical service producer-consumer network topology](image)

Figure 2.1: Typical service producer-consumer network topology

עוג 2.1: תיאור שרשרת שירותים עם אגף שירותים שיווקי

9
One of the goals of a monitoring system is the ability to perform quick configuration updates without changing the service provider or the service consumer. Enabling or disabling the monitoring for a specific service, or changes in the policy for a service should not be accompanied with any changes to the parties. SLAMES fulfills most of this requirement by being transparent as much as possible to both sides. When using SLAMES in a standard way, the service provider does not need to know about SLAMES except for security reasons. SLAMES behaves as a service consumer to the real service provider, and as a service provider to the real service consumer. This behavior is a typical use of the proxy design pattern. All monitored communication passes through SLAMES in its way from the service consumer to the provider and vice versa.

The service consumer should be aware of SLAMES, since it is the side who initiates the communication and needs to send the service requests to SLAMES instead of the real service provider. However, that is the only change needed in the service consumer. In order to reduce the changes in the service consumer to the minimal level, SLAMES provides the same Web service interface as the real service provider, meaning that the only difference between using SLAMES and using the real service provider is the URL of the service. Some Web services frameworks, such as .Net XML Web services and some JavaEE Application Servers, provide standard interception mechanisms that can change the URL of the service via configuration files without making any changes to the application code. Hence, when operating in these frameworks, SLAMES can be made fully transparent to the service consumer application code.

Upon disabling the monitoring for a specific service on SLAMES, the system will continue to function as a regular proxy to the service provider without making any changes in the service consumer. This involves an expected performance overhead for using SLAMES as a proxy instead of connecting to the real service provider directly.
2.2 The Proxy-based Approach

The rationale behind using a proxy-based approach for monitoring web services was well-explained in the previous section: (i) Using the same interface as the original Web service interface provides an abstraction layer that can be used to solve many problems including network performance issues with minimal changes to the service consumers. As illustrated in Figure 2.3, the system is physically in the network boundaries of the consumer, while logically part of the service being offered by the provider. (ii) The proxy abstraction layer provides capabilities for quick changes in the configuration, allowing to attach or detach SLA requirements to existing services even without the support of the service provider. (iii) Being a third party in this world of services, it is an opportunity for firms working with network infrastructures and having a lot of expertise in this field to have their contribution for service performance in some scenarios when the service providers cannot control or guarantee.

However, we also notice potential problems with this approach:

1. Since SLA-based information is tracked only in our system, we cannot detect problems that occur between the Web service consumer and SLAMES.

2. SLAMES itself provides a performance overhead, thus affecting the actual quality of the service level provided to the service consumer.

3. Such an abstraction layer encourages a centralized implementation on one host, and as such, may function as a performance bottleneck and a single point of failure for all the monitored services.
4. Some performance problems are caused by the service provider and not by the network environment. Such problems are impossible to solve without the intervention of the service provider.

An intelligent design of the system should minimize most of the problems. In order to solve the first problem, we should install SLAMES in a location close as much as possible to the service consumer. It is recommended to install it on the same machine as the service consumer itself if possible. However, a host with a direct local area network access to the client is also a good compromise.

In order to minimize performance overhead, we should provide a scalable implementation. For instance, we can use several SLAMES instances in various configurations - an instance for each consumer, an instance for each monitored Web service, a pool of instances for load balancing of an intensively-used Web service, etc. SLAMES is built with scalability issues in mind in order to address these problems and minimize them as much as possible.

Performance problems caused as a result of a load on the service provider cannot be solved by SLAMES. However, SLAMES can cooperate with a service provider in order to alert the service provider when it is suspected to be the source of the performance problem and request the provider to try and fix this problem.

2.3 High-Level Architecture

Figure 2.4 shows the high-level architecture of SLAMES, which consists of three components:

1. A persistency layer for keeping information about the registered services and the information to track about them.

2. SLAMES engine, which handles the service requests for services registered on this server, monitors the required performance parameters and responds to SLA violations according to the defined policy.

3. A management Web service for registration of services in the system and configuration of policies for the registered services.

As shown in Figure 2.4, there is no direct connection between the management Web service and the engine itself. It is the responsibility of the persistency layer to share and synchronize the configuration data between the engine and the manager. This design is optimized for the scalability of the system. Scalability issues will be addressed later in this work.

One of the key features in SLAMES is its powerful policy capabilities. SLAMES policy definition uses a scripting language that can be used to define sophisticated conditions for a policy violation and write a flexible response code when such a violation occurs. We believe that policy configuration should be done by experts in this field and therefore they should be given a maximum level of power and flexibility.
2.4 Use Cases

In this section, we show some real world scenarios where SLAMES can be used to effectively monitor and solve performance violations.

Consider a case of an investment agent traveling from one place to another. He needs to be constantly updated with the current stock market and share prices. Connection loss or unexpected long network delays might prevent the agent from doing urgent operations in the stock market, and consequently to lose money. Since his location is always changing, he uses a wireless connection with access to the stock market through a mobile ad-hoc network (MANET). With SLAMES integrated with his environment, the agent can monitor long response times caused by the dynamic characteristic of these networks, and try to solve these problems by re-invoking DSR (Dynamic Source Routing, a network path resolution algorithm) in the network.

SLAMES can also be used to help an Internet TV customer. In order to let a customer watch video channel over the Internet with good quality and smooth operation, a sufficient bandwidth should be guaranteed to the customer. However, while the TV service provider can guarantee its bandwidth to the customer, there is usually no guarantee on the bandwidth in the network environment. Therefore, the customer might not get the promised bandwidth due to network constraints. SLAMES can be used in such a situation in cooperation with the service provider itself. Whenever the promised bandwidth is not provided to the customer, SLAMES detects the bandwidth violation and communicates with the service provider in order to change the mirror from which the streaming content is consumed.

Another common scenario is the case of an electronic shop using a third party payment service. The availability of such service is crucial to the electronic shop. SLAMES can be used to ensure that the payment service is available for use most of the time, and depending on the defined policy, to contact another payment service if the availability of the current payment service is getting lower than a given threshold.
Chapter 3

SLAMES Policy Language

This chapter describes the syntax and the language of SLAMES policies. SLAMES policy is a mixture of XML-based syntax [31] and a Python-like [27] scripting language. These two extremely different languages were merged in order to provide a simple yet powerful policy definition, that is also compatible with the WS-Policy standard [35]. SLAMES policies support specifying constraints on performance and availability aspects of web services. However, they can easily be extended to support other aspects such as security and reliability as well. Since such aspects are not the focus of this study and have their own accepted standard, we do not describe them here under this policy. Moreover, SLAMES policy can be combined with these standards by using WS-Policy aggregation capabilities.

The contents of a SLAMES policy have a schema-based XML structure. Some elements in the schema are more complicated than regular XML elements and contain a scripting code written in Jython, a Java-based implementation of the Python programming language [14]. A complete definition of the SLAMES policy schema is specified at the end of this thesis.

Figure 3.1 presents the general structure of a SLAMES policy. The top-level element of a SLAMES policy is the <policy> element. This element is the SLAMES-specific definition of a Policy Assertion, as defined in the WS-Policy standard. Since SLAMES policies do not only declare assertions, but also specify how to handle violations of the assertions, the behavior in cases where we aggregate more than one SLAMES policy using WS-Policy’s <All> & <ExactlyOne> elements is undefined. To simplify the evaluation of the policy and for a clear definition of the desired policy behavior, aggregation of SLAMES policy is allowed only as a single SLAMES <policy> element under a top-level <All> element. Aggregation to other policy definitions, such as WS-Security or WS-Trust is permitted under this constraint. In addition, the evaluation of a policy assertion with an <All> element that contains a SLAMES policy should apply the SLAMES policy only after all other assertions under the <All> element are verified and applied.

Figure 3.2 demonstrates a valid WS-Policy definition that combines a SLAMES policy with security constraints according to the WS-Security standard [37]. The policy uses the SLAMES policy along with an encryption for the SOAP messages. The allowed encryption algorithms are either Basic256Rsa15 or TripleDesRsa15.

The <policy> element contains one or more <event> elements, each of which represents a possible violation event in the policy. Each <event> contains a pair of two elements - a
condition and an action. The condition specifies when a violation occurs, and the action defines the desired behavior of the system in that case. An optional `<configuration>` element contains several configuration settings for the specified policy.

### 3.1 Policy Conditions

As shown in Figure 3.1, SLAMES supports two condition types: an event-based condition and a time-based condition. An event-based condition, declared using a `<condition>` element, is simple and more efficient than time-based condition. It provides a set of assertions for predefined monitored performance variables such as latency, throughput, etc. A time-based condition, declared using a `<timed-condition>` element, is a complicated condition written in a scripting language and evaluated at constant time intervals. SLAMES currently supports only Jython as a scripting language, but has the option of using other scripting languages as well.

Figure 3.3 describes an example of an event-based condition. According to this example, if a service request call takes longer than 30 seconds, written as 30000 milliseconds, and the service received an average of more than 100 requests per second and the response rate per second is also higher than 100, then SLAMES should indicate that there is a violation for
Figure 3.2: A Valid WS-Policy Definition with SLAMES

this condition and the associated action should be invoked. Latency, throughput and load are examples of three variables monitored and evaluated during SLAMES operation.

Figure 3.4 specifies the same condition given in the previous event-based condition example, but this time with a time-based condition. There are, however, some differences between the first event-based condition and this condition:

1. The first condition is checked only when the values of latency, throughput and load variables are changed or reach the specified thresholds, while the second condition is evaluated regardless of the involved variables every one second (1000 milliseconds).

2. The first condition is checked using SLAMES internal representation of monitored values. Thus, its evaluation is considerably fast. In the second condition, the Jython scripting interpreter engine evaluates the result of the condition, which adds an extra overhead during the process, making it much slower than evaluating the first condition.

In these two condition examples, we used three variables whose value is calculated by the SLAMES core. SLAMES provides several more variables for giving more flexibility to the policy writer. Some of the variables are statistical computation of a monitored value over a time period. This time period, also known as History Period, is configured under the optional `<configuration>` element. If no configuration is defined, SLAMES uses some
default values. The variables, their types and their exact definitions are explained in Table 3.1.

The variables keeping a history array of values cannot be used within an event-based condition since it makes less sense to query a specific value in them. However, they can be used in a time-based condition or in any action definition.

In addition to the variables declared in Table 3.1, there are some internal variables that are used to calculate other monitored values. For instance, the service request timestamp is kept in the variable requestStartTime in order to support the calculation of a service request latency. It is possible to query the values of these variables in time-based conditions or in actions, although their value is not likely to be useful.

It is important to notice that event-based conditions are evaluated as a result of a specific service request event while timed-based conditions are not. It means that time-based conditions cannot access request-associated variables, such as latency, since the condition is not associated with a specific request whose latency should be compared to. In this case, it is reasonable to query the value of an aggregation variable like averageLatency, which is not associated with a specific request.

3.2 Policy Actions

The <slm:action> element represents the action to be invoked when the associated condition is satisfied. Actions are identified by their types. They can use an optional string parameter specified as the contents of this XML element. Figure 3.5 declares two actions. The first one is a predefined action that initiates a new discovery of the network path to the service provider using a routing algorithm. The second example uses an action called 'jython' which runs the contents of the element inside a jython interpreter. In this example, the action prints the string ‘action was called!’.

SLAMES provides an alternative way for specifying the contents of an action. When defining an action with the optional file attribute, the contents of the action will be read from the specified file instead of the contents of the <action> element. It is primarily useful for associating the same action script code with different events in the same or in different policies.

SLAMES actions have powerful capabilities. An action can access all the monitored variables, query for their values, change their values and even add new variables that can be used by other actions. Actions are also given some control on SLAMES operation using Control Variables. These variables provide the action with some API to change things in
### Table 3.1: SLAMES Monitored Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Variable Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency</td>
<td>integer</td>
<td>The number of milliseconds for a specific service request to return a result.</td>
</tr>
<tr>
<td>averageLatency</td>
<td>float</td>
<td>The average value of ‘latency’ during the history period.</td>
</tr>
<tr>
<td>load</td>
<td>float</td>
<td>The average number of service requests per history time unit during the history period.</td>
</tr>
<tr>
<td>throughput</td>
<td>float</td>
<td>The average number of service responses for requests per history time unit during the history period.</td>
</tr>
<tr>
<td>availability</td>
<td>float</td>
<td>The percentage of service requests during the history period whose response was received. The value is in the range [0,1].</td>
</tr>
<tr>
<td>latencyHistory</td>
<td>array of integers</td>
<td>a history-array of the recent latency values during the history period.</td>
</tr>
<tr>
<td>loadHistory</td>
<td>array of integers</td>
<td>a history-array for the number of service requests during the history period.</td>
</tr>
<tr>
<td>throughputHistory</td>
<td>array of integers</td>
<td>a history-array for the number of service responses during the history period.</td>
</tr>
</tbody>
</table>

1 A history-array is an array that contains the recent values of some monitored variable during the history period. The values are sorted according to their monitored time, while the first element in the array is the last monitored value.

The operation of SLAMES regarding the monitored service or the specific service request. The supported control variables and their capabilities are specified in Table 3.2.

When a jython action is initiated, all official monitored variables and control variables are evaluated and assigned to variables in the action jython environment. These variables contain only the values of the monitored variables and changing them will not modify the real monitored variables. In addition, a variable called state contains a mapping between monitored variable names and their wrapper objects. Accessing monitored variables using this variable enables both read/write operations to the real value of the variable. Accessing control variables using the state variable enables us to use the control API for controlling SLAMES.

Since actions are associated with conditions, there are differences between the variables available to them. Similarly to time-based conditions, actions associated with time-based conditions cannot access request-associated variables. These include the request-associated control variables, like httpClient and httpMethod. Instead, they should access only aggregation variables that count average and extreme behavior.
3.3 Policy Configuration

SLAMES policy definition supports an optional configuration section. This section contains definitions of some values used by the monitoring and enforcement components. Figure 3.6 provides an example of a configuration. Policies with this configuration will keep a latency history of one minute, written as 60000 milliseconds, and a load history for the count of service requests for each minute in the last 60 minutes. All configuration parameters are specified in Table 3.3.
**Table 3.2: SLAMES Control Variables**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>policyManager</td>
<td>provides control over the URL resolver and an option to invalidate the current policy. The URL resolver is used for getting the real URL of service provider. Changing the URL resolver enables actions to change the real address that requests are forwarded to. For instance, actions can define some replicas to the service provider and use them upon failure or in a round-robin strategy in order to provide load-balancing support. Invalidating the policy is useful when the action changes the policy associated with this monitored service. This action will unload the policy, delete all service’s collected statistics, and reload the policy again once a service request to the appropriate monitored service is received.</td>
</tr>
<tr>
<td>httpClient</td>
<td>request-associated variable that provides an option to abort a pending request to the real service provider. It is also possible to use it in order to initiate a new service request instead of the aborted one.</td>
</tr>
<tr>
<td>httpMethod</td>
<td>request-associated variable that provides an access to the actual parameters of the request forwarded to the real service provider.</td>
</tr>
</tbody>
</table>

**Table 3.3: SLAMES Configuration Parameters**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency-history-millis</td>
<td>integer</td>
<td>The size of latency history in milliseconds. Used by the variables latencyHistory and averageLatency.</td>
</tr>
<tr>
<td>load-history-size</td>
<td>time period with unit(^1)</td>
<td>The period and time unit granularity of service load monitoring. Used by the variables loadHistory and load.</td>
</tr>
<tr>
<td>throughput-history-size</td>
<td>time period with unit(^1)</td>
<td>The period and time unit granularity of service throughput monitoring. Used by the variables throughputHistory and throughput.</td>
</tr>
<tr>
<td>availability-period-minutes</td>
<td>integer</td>
<td>The period of availability history statistics in minutes. Used by the availability variable.</td>
</tr>
<tr>
<td>no-violation-timeout-minutes</td>
<td>integer</td>
<td>The number of minutes to wait between triggering the actions of the same policy event during a continuous violation of the policy condition, as long as the violation was not solved during this period.</td>
</tr>
</tbody>
</table>

\(^1\)This kind of a value is a combination of a positive integer specifying the size of the collected history, and an extra letter specifying the time unit: s - for seconds, m - for minutes, h - for hours and d - for days. A value of ‘10h’ means that the history is kept as a count of all relevant events for each hour in the recent 10 hours.
Chapter 4

SLAMES Implementation

Chapter 2 explained the high-level architecture of SLAMES. In this chapter, we discuss the implementation of SLAMES in more detail - its components and the technologies used for the implementation of each part. We explain how to scale up this system in order to monitor more services while maintaining a small performance overhead. We also mention how to extend SLAMES in some aspects in order to add more functionality and which optimizations should be applied in order for SLAMES to be used in some tight time-critical environments.

SLAMES runs on top of Apache Tomcat server [2], an open-source lightweight container for Web applications. Tomcat is known as a fast server with a small memory footprint, but it does not support the full functionality of a JavaEE (Java Enterprise Edition) application server. In order to work with Web services, we need to deploy a Web services runtime engine on Tomcat. We chose to use the open-source Apache Axis2 library [1] as our runtime engine for handling Web services.

4.1 The Engine

SLAMES engine is the core of the system. This component is responsible for the primary functionality of the system - monitoring Web services performance using the proxy-based approach. The engine receives all Web services’ call invocations and sends them to the real service provider while monitoring their performance and responding to SLA violations. It is implemented as a Java Servlet using Sun Microsystems JavaEE Servlet technology [29].

The naive approach for building a monitoring proxy for a Web service is using code of existing frameworks, such as Apache’s Axis, Apache’s CXF, Spring or .Net web-service infrastructure in order to do most of the work. Most Web services frameworks provide a facility for client code generation and skeleton server code generation given a WSDL document representing the interface of the Web service. The open-source Apache Axis library, for instance, has that kind of capabilities and can be used in order to provide an API that generates a proxy for a given Web service.

An implementation of such an API uses the framework to generate code for the Web service client that connects to the real service, and also generates a Web service provider implementation code that wraps the generated client code along with specific code for
monitoring the performance. Once the proxy is generated, it should be compiled and deployed on an application server and can be used as a single Web service proxy. This proxy should be generated for each monitored Web service.

There are several problems with this kind of a proxy mechanism:

1. It generates a lot of classes for each monitored Web service. A generated code that has to be compiled and deployed suffers from robustness problems and it is very hard to maintain.

2. Several Web service runtimes, such as Apache Axis, do not allow to deploy a new Web service without restarting the server. We cannot allow a server restart each time we want to monitor a new Web service since the other monitored Web service proxies are unavailable during that operation.

3. Changes in the interface of a Web service or changes in the SLA requirements of a specific Web service will force a re-generation of the Web service proxy code from scratch, with a small downtime.

These problems led us to choose a different model for implementing the proxy-based approach, using a single servlet that is serving Web service invocations from all service consumers. It identifies the appropriate service according to the path within the URL of the request, the same way RESTful Web services [9] use the URL to identify an object. Then, it finds the policy associated with this service, and forwards the request to the real service provider while monitoring the invocation. When a response is received from the real service provider, it updates performance statistics and sends the response to the service consumer.

The servlet implementation does not use code generation at all nor need any on-the-fly deployment capabilities. Services can be dynamically added or removed during the operation of the servlet with no need to restart. The implementation is much simpler, and therefore easier to maintain.

The drawback of the servlet approach is that there is a single proxy servlet to all services. This is in contrast to the first approach, in which a separate proxy is generated for each service through code generation. Hence, it is more complicated to make a servlet scalable than several non-dependent Web service proxies.

One of the interesting issues in the SLAMES engine is the lazy nature of the servlet. The application server does not initialize nor load the servlet until it is really needed to handle a consumer’s Web service call invocation. Moreover, the servlet internal implementation has a lazy behavior, so that registered services are not monitored if they are not in use at all unless an active monitoring constraint was set, such as checking the availability of a service. This way, unused services do not add an extra unwanted overhead on the performance of the servlet and the memory footprint of the servlet is much smaller.

4.2 The Management API

The management API of SLAMES is implemented as a Web service. It enables configuration changes with operations for registration and removal of monitored services and change of
the policies associated with the monitored services. It is possible to register a service more than once in SLAMES and associate a different policy for each registration. This way, the same service can have different levels of monitoring according to different consumer needs.

The management Web service updates the configuration stored by the persistency storage using the persistency layer. It does not communicate with the engine directly. Configuration changes are updated in the engine by the persistency layer.

When using the management API to register a new service for monitoring, the management API provides a relative path to the SLAMES server that along with the SLAMES server address represents the URL of the monitored service in SLAMES. We also retrieve the WSDL document of the monitored service. This WSDL document contains the definition of the service interface that is the same as the original one, and also contains the policy associated with this monitored service. Using the URL of the monitored service, we can retrieve its appropriate WSDL document with the associated policy.

4.3 Persistency Layer

The persistency layer is responsible for the storage of all data saved in SLAMES that should be kept even when the system is down. This includes all the monitored services, the details about the real service providers and all the policies associated with the monitored services.

One of the goals of the persistency layer is to hide the real storage implementation. Both the engine and the management API use a common interface for storing and retrieving data and do not care whether the persistency is taken care using files, database or any other technology.

The exact implementation of the persistence layer is transparent to the rest of the proxy architecture. It can be easily replaced with another implementation in order to match different performance requirements. For the evaluation of SLAMES performance, we used two different implementations for the persistency layer: (i) A file-based implementation, which stores the configuration of SLAMES in a file on the local file system. (ii) A database implementation, which stores the configuration of SLAMES in a local database repository. We use Sun Microsystems’ MySQL database for this implementation.

The file-based implementation has one important advantage over the DB implementation. It is very lightweight since it stores all the information in memory and it does not require any additional dependent database framework. We tend to use this implementation when a small overhead for SLAMES is more important than the scalability of the system, or when a database environment cannot be installed in SLAMES’ environment. The file-based implementation does not support synchronization among multiple machines, and therefore it is very limited in its scalability support. It can be extended to support multiple machines, but such usage is not recommended.

The database implementation is expected to have a higher overhead in comparison to the file-based implementation. However, since most database queries in SLAMES are read operations for the same values, and as modern DB driver engines use cache mechanisms to keep the results of the last queries, the extra overhead of using a database in SLAMES insignificant as seen in our measurements. This implementation is preferred over files when SLAMES runs in a multiple machine environment, since DB implementations solve a lot of
synchronization issues. By using a DB, we need not reinvent the wheel on these well studied issues. Most DB implementations even support clustering of the database server over several machines for scalability. In addition, application server implementations provide convenient methods for working with DB and configuration of DB settings in the application server itself, instead of putting those configuration details inside the persistence layer implementation.

4.4 Scalability Issues

A lot of thought has been spent in the design of SLAMES in order to let it scale well. We mentioned earlier several design and technology choices that are used for making SLAMES more scalable. In this section, we explain how can we use several machines in order to handle a larger amount of monitored services while trying to minimize the overhead involved with a multiple-machine environment.

SLAMES is implemented using two technologies that already have solutions for scalability problems. Using a database for the persistency layer is crucial for scalability. Modern database implementations support synchronization among clients connecting from different machines, and also support the installation of the database on a cluster of machines in order to provide load balancing and reduce database overheads.

SLAMES uses the standard capabilities required from a JavaEE Application Server as defined in the standard. There are many available implementations of JavaEE application servers (or just Web Containers) that support a cluster environment. Popular enterprise application server implementations include IBM Websphere Application Server, Oracle Application Server, Oracle Weblogic Application Server, Sun Application Server, Red Hat JBoss, etc. These application servers have a proven clustering capabilities used in time-critical environments. Even the lightweight Apache Tomcat supports a cluster configuration. SLAMES can be easily deployed on top of these middleware technologies in order to use their scalable implementation.

In addition to these technologies, several design decisions were made in order to ease the installation in a cluster environment. Since the engine and the management API interact only using the persistency layer, they can be deployed on different machines. It is also possible to install several instances of these components on different machines. The persistency layer will take care of the synchronization among those instances.

However, an extra care should be taken when running the engine servlet on several machines. Since the performance information collected on a monitored service is not persistent in our implementation, we should not monitor the same service on more than one machine. Web service requests from a specific monitored service should be redirected to the only machine monitoring this service by the load balancer using the URL of the request, which specifically identifies the monitored service. It is also possible to support monitoring of a specific service on several machines by extending SLAMES to support persistency of the collected performance information. Such capability is not hard to implement, but we expect it to add a non-negligible performance overhead.

Upon implementing a cluster with no load balancer, SLAMES can support degenerate use of several machines by providing different hosts in the URL of the monitored service when registering a new service to monitor. Such a “load balancing” scheme statically assigns
a monitored service to a specific host regardless of the dynamic load on that machine. It is not recommended, but is still useful in cases when a real load balancer cannot be used.

Another simple yet useful scheme for distributing SLAMES over several machines is to install SLAMES as a standalone system on each machine, along with its own persistency storage. Since the current implementation uses relatively lightweight implementations for a database and an application server, it is possible to install each SLAMES on the consumer host as long as the same service does not need to be monitored for several consumers. This solution is highly encouraged in dynamic networks like MANETs where it is not trivial to establish a reliable multi-machine environment close to the service consumer.

4.5 Engine Extensions

When designing SLAMES, the importance of the ability to extend it with new functionality was well understood. In this section, we describe the steps needed in order to extend SLAMES in some directions that we thought SLAMES is more likely to be enriched with additional functionality.

4.5.1 Adding a new action

In order to support new actions to be executed upon SLA violations, the extension writer should implement a class for the action and register the action within the action registry. A new action class implementing the action interface should provide a unique action name that is used within the policy in order to refer to that action in the type attribute of the <action> element. In addition, the writer has to specify which monitored variables are used in the implementation. It is needed in order to support monitored value optimization, so that unused variables may not be monitored in some cases when they are not needed. The action is given the contents of the policy <action> element as an optional parameter to the action.

4.5.2 Adding monitored variables

Extending SLAMES to support new types of monitored variables consists of two stages: creating a variable class that stores the collected information and creating a monitor class that updates the value of the variables on some SLAMES events.

A variable class implementation must provide a method for retrieving the value of the variable and possibly one or more methods for affecting the variable value. SLAMES is not obligated to access variables from a single thread. Hence, it is the responsibility of the variable class implementation to synchronize the access so that the variable class is thread-safe.

In addition to the variable implementation, the extension writer should annotate the variable class with a variable descriptor in order to specify its name, its condition reader (if one exists) and whether or not it is a request-associated variable. Then, the variable should be registered in the Variables registry.
Specifying a variable descriptor and an implementation helps to maintain its value, but does not specify how and when to calculate it. In order to properly update the variable value, the extension writer should implement a monitor class that receives the events of request handling start, finish and failure. Then, this monitor class should be registered in the policy enforcer, which exists for each monitored service.

A variable can be created and added to the state of the monitored service from a Jython script without being updated in any monitor implementation. In such cases, the policy should update the variable value from the same or other Jython scripts.

A typical monitor updates the value of several related variables. For instance, the variables latency, latencyHistory and averageLatency are updated in the same monitor since they use each other’s values. SLAMES does not support monitor dependencies, which force a monitor to be independent with respect to other monitors. It means that a monitor should not use variables updated by other monitors or declare any variables that use variables of other monitors. This is because other monitors are not guaranteed to be loaded unless there is a policy that uses values of their maintained variables.

### 4.5.3 Adding event-based conditions

Event-based conditions are usually used for evaluating a condition about a monitored variable, although there is no restriction to use a single variable value.

The extension writer should implement a condition class that specifies which variable values are used in the condition and calculates whether or not the condition is satisfied given the variable values.

In addition, the extension writer should implement a reader class that parses the condition from the policy XML document with all the relevant parameters. This reader class should be added to the variable descriptor of one of its used variables in order to register it in the condition reader registry.

The policy element of the new condition is the name of the variable used for registering the condition. In order to use a different name for the element of the condition, one should create a dummy variable with the desired element name as the variable name in its descriptor and use that variable in order to register the condition reader.

### 4.6 Future Engine Optimizations

SLAMES engine applies some optimizations for reducing the memory footprint and using as little resources as possible during its operation. This approach is used in the lazy policy loading and monitor loading schemes: policies are not loaded unless their associated monitored service was sent a request, and monitors maintaining values that are not used in a policy are not loaded at all when enforcing the policy of a monitored service. However, there are several changes and optimizations that are needed if one is to use SLAMES in a time-critical scenario or in a highly-available environment.

In a time-critical scenario, we would like to put some limit on the amount of CPU time spent on evaluating conditions and execution violation actions. Since both conditions and actions can be implemented using Jython scripts, a poor implementation may cause
undesired consequences, such as stopping or delaying the enforcement of a specific monitored service. In order to prevent such scenarios, we would like to put that limit in our monitoring system. In order to better preserve the fairness of the CPU time consumption among the monitored services, a script that runs for a long time should be given less CPU time. Such a mechanism is similar to process scheduling schemes in modern operating systems.

Another optimization can be employed to the calculation of monitored aggregation variables, such as $\text{averageLatency}$ or $\text{load}$. The current implementation of these variables in SLAMES do not store the aggregated value in the variable, but re-calculates them with each retrieval of their value. This heuristics is good when the variable values are used only a few times while the variables they depend on are not changed. A different approach may be to calculate and store the values of these variables whenever their dependent variables are changed. This approach may perform unnecessary calculations if these variables are changing faster than their value retrieval rate. We recommend on a compromise method between these approaches: use a validation flag to mark when an aggregation variable value is no longer valid due to changes in its dependent variables. Upon value retrieval, the variable uses its last calculated value if it is still valid according to the validation flag. Otherwise, the variable should re-calculate its value and store it for later use.

In a highly-available environment, we would like to ensure that SLAMES is always up and running. In order to ensure that, we can replicate SLAMES engine over several machines, as specified in the scalability section of this chapter. However, if one of the machines crashes, SLAMES loses all the statistics gathered about the monitored services on that machine. In order to prevent this information loss, we need to add the collected statistics to the persistency layer of SLAMES, so that recovering from a crash will restore the collected statistics of the monitored services. We expect such an addition to the persistency layer to cause a much higher performance overhead for SLAMES. We suggest to store the collected statistics as a background activity once in a predefined time period, so that statistics storage will not be on the critical execution path of the engine proxy mechanism, and yet the statistics will be kept in the persistency layer, even if it is not the most recent state of these statistics.

4.7 Class Diagram

SLAMES implementation contains about 100 classes and interfaces. Figure 4.1 describes the relationship among the primary classes in each of SLAMES components according the UML standard notation.
Figure 4.1: The Primary Classes in SLAMES

Legend

- A ⚫ B  A depends on/uses B
- A ◦ B  A contains an instance of B
- A ◬ B  A contains several instances of B
- A ▲ B  A inherits/implements B

Technion - Computer Science Department - M.Sc. Thesis
MSC-2009-06 - 2009
Chapter 5

Performance Evaluation

5.1 Evaluation Goals

This chapter presents the performance benchmarks of SLAMES in order to better understand the implications of using SLAMES in our environment. We would like to know how much does it cost to use SLAMES in terms of response time overhead, and realize what is the preferred configuration of SLAMES given a specific scenario.

For this purpose, we tested SLAMES performance with policies expected to incur different levels of response time overheads, measured the effect of monitoring several Web services on a specific service and evaluated the differences among several configurations, such as local vs. remote installation or file vs. database based persistency.

5.2 Evaluation Environment

5.2.1 Machine Setup

We used 4 machines in order to evaluate SLAMES: one host represents the Web service provider, one host contains the installation of SLAMES, and two hosts were used as Web service consumers in different configurations, according to the tested scenario.

All machines share a Gigabit network connection and have the following hardware & software specifications:

- CPU: Intel Core 2 Duo, Model T2140 (2GHz, 533MHz FSB, 1MB Level 2 Cache).
- Memory: 2048 MB at 667 MHz.
- Network Interface: BroadCom NetLink Gigabit Adapter.
- Java Virtual Machine: Sun Microsystems JDK 6 update 10 (version 1.6.0_10).
- Apache Tomcat 6.0.18 with Apache Axis2 1.4.1.
• Sun Microsystems MySQL 5.1 installed locally on SLAMES machine (used only in database performance evaluation).

The calculator Web service provider is using Apache Tomcat with Apache Axis2 library for supporting Web services. The Web service consumers use Apache Axis2 generated code for Web service client in order to invoke the Web service calls.

### 5.2.2 Evaluated Policies

We evaluated the performance overhead of SLAMES using 4 monitored services with different associated policies:

1. No policy - using SLAMES as a regular Web service proxy.
2. Throughout-based policy - a minimal policy with a simple condition on the *throughput* variable, whose evaluation is not intensive for the system.
3. Latency-based policy - a policy that includes two simple conditions on the *latency* variable, whose evaluation is expected to be more intensive than *throughput* variable. In addition, one of the conditions is associated with a Jython action, which is more intensive than a regular Java action.
4. Intensive policy - a policy with 3 conditions that use almost all monitored variables. One of the conditions is time-based written as a Jython expression. In addition, one of the actions is a Jython action.

The full definitions of the policies used in the performance evaluations are specified in Appendix B.

### 5.2.3 Evaluated Web Services

In order to evaluate SLAMES performance, we used a simple calculator Web service with 4 operations: add, subtract, multiply and divide.

The rationale behind choosing a simple Web service is to minimize the Web service execution time in order to measure the message parsing time and the network delays. A simple fast Web service is also useful for generating an intensive Web service activity, so we could measure the performance under high operation loads.

### 5.3 Performance Evaluation Results

#### 5.3.1 Performance of Different Policies

The most important factor for evaluating the performance of SLAMES is its overhead on the performance of the monitored services. We would also like to know what is the cost of using SLAMES as a regular proxy with no policy and how much performance degradation do we suffer when using intensive policies.
In order to answer these questions, we first evaluated the performance of a direct connection to the Web service provider. Then, we evaluated the performance for each of the policies described earlier.

Since the evaluated Web service has no complicated calculations to perform and the network connection is a reliable Gigabit connection, we expect that the response time of the evaluated Web service using a direct connection is the time required to parse the SOAP messages on both the service consumer and the service provider.

Each policy was evaluated after a clean SLAMES startup in order to eliminate any performance effect of previously used monitored services in the running SLAMES instance. Each policy was tested with a varying number of concurrent Web service calls, which were invoked by two Web service consumer hosts in several threads.

Each benchmark result was calculated as the average of 1700 Web service calls. Since the accuracy of our time measurements is $\sim 16\text{ms}$, this is enough for accuracy of an $0.01\text{ms}$ for each result. For each combination of a policy and number of concurrent calls, we measured several benchmark results and calculated their average. Since the thread scheduling is under the JVM control, we cannot assure that all Web service calls start exactly at the same time, and we cannot assure they finish together. Hence, the actual number of concurrent calls in these cases is not exactly the same as the one we officially intend to measure. In order to correctly measure the performance, we removed these results when calculating the average result. We also removed results influenced by sporadic network or operating systems delays and deviated more than 100% from the average result.

![Average Response Time of Different Policies](image)

**Figure 5.1: Different Policies Evaluation**

зиיפד 5.1: העריכה במתחים של הקטיפות השעון
<table>
<thead>
<tr>
<th>Evaluated Policy</th>
<th>Average Overhead (ms)</th>
<th>Average Overhead Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>No policy (proxy)</td>
<td>2.692</td>
<td>1.488</td>
</tr>
<tr>
<td>Throughput-based policy</td>
<td>2.802</td>
<td>1.507</td>
</tr>
<tr>
<td>Latency-based policy</td>
<td>2.738</td>
<td>1.493</td>
</tr>
<tr>
<td>Intensive policy</td>
<td>2.967</td>
<td>1.533</td>
</tr>
</tbody>
</table>

Table 5.1: Average Overhead of Evaluated Policies

Figure 5.1 shows the response time of the different policies in comparison with direct Web service calls. The graph shows that the overhead of using SLAMES as a proxy is quite similar to the overhead of using it with the considerable lightweight policies. As expected, the intensive policy has a higher overhead. However, this overhead is still not far from the overhead of the other policies. According to Table 5.1, the average overhead of SLAMES for all policies is around 50% of the time it takes to parse SOAP messages in our environment. If we consider real network delays as well as realistic execution time of Web services, then the relative part of SLAMES in the total response time of the Web service is significantly lowered. We should also notice that our benchmarks generated continuous immediate Web service calls that are different from the Web services common usage rate.

5.3.2 The Performance Effect of Other Monitored Services

Another important aspect of SLAMES performance is how much does a given monitored service affects the performance of other services monitored in the same instance of SLAMES. In order to evaluate this effect, we measured the average response time of the throughput-based policy while doing some background Web service calls.

The benchmark tests use 3 different strategies for generating background Web service activity:

1. Low rate random policy - each background activity thread invokes a monitored Web service call with a random policy among the throughput-based, the latency-based and the intensive policies. The thread waits a random period of $10 - 100\text{ms}$ between two subsequent Web service calls.

2. Frequent random policy - same as the low rate random policy, but the thread does not wait any time between frequent Web service calls.

3. High rate random policy - each background activity thread chooses a random policy at the beginning of the scenario. This policy is used for the rest of the scenario for generating frequent Web service calls. Since the same policy is used by that thread, there is no need to open and close network connections for each call. Hence, the invocation rate is much higher.

Each of the specified strategies was compared with varying number of concurrent background threads to the performance of the single thread throughput-based policy with no background service calls.
As shown in Figure 5.2, the effect of other monitored services on the performance of a single monitored service is minor when SLAMES is not under high load of Web service calls. A major performance degradation was measured only in the High rate random policy strategy, which we consider as an extreme usage of Web services. Moreover, we can see that the overhead of other monitored services on the performance of our single monitored service is not higher than the overheads measured in the previous section when we evaluated the overhead of concurrent Web service calls that use the same policy. Hence, we conclude that the primary factor on the performance degradation is the number of concurrent Web service calls using SLAMES, while the policies used in these Web service calls have only a secondary impact.

### 5.3.3 Local vs. Remote Installation

When describing the architecture of SLAMES in Chapter 2, we proposed two topologies for using SLAMES: local installation on the same machine of the Web service consumer, or remote installation in the local area network of the Web service consumer. In order to realize the effect of the chosen topology on the performance of SLAMES, we compared the performance of both topologies when using SLAMES as a proxy with no associated policy. This is the optimal scenario for isolating the network delays caused by the remote installation. We should also notice that while remote installation have expected network delays, the local installation suffers from double rate of networking I/O system calls and message...
processing that also come with an expected performance cost. Unlike previous benchmarks, in this benchmark we did not remove the results that suffer from poor performance due to unexpected network peaks since network delays are the measured part in this scenario.

Figure 5.3 shows the response times of concurrent Web service calls with local and remote installations. We see that the local installation is faster when the number of concurrent calls is relatively low. But in much higher loads, message processing time and operating system networking overhead become more dominant, and using a remote installation performs better in our Gigabit reliable local area network.

5.3.4 File vs. Database Based Persistency

SLAMES provides two implementations of the persistency layer: a file-based implementation and an implementation using a database. The file-based implementation reads all the persistent data from the file system to the memory upon SLAMES startup, while saving the state upon each change. Using a database for implementing the persistency layer reads a specific data from the database upon request, and updates the database whenever there is a change in the monitored services.

Since changes in the monitored services are rare in respect to the number of the monitored Web service calls, we are not interested in evaluating the performance of the management API in both implementations. We are only interested in the performance overhead.
of these implementations during the ongoing operation of SLAMES engine handling monitored Web service calls. We used the throughput-based policy for evaluation of the file-based versus the database persistency performance.

We expected database persistency to be much slower than the file-based persistency. However, as can be seen in Figure 5.4, both implementations have similar performance for monitored service response time. The reason for this behavior is that most SLAMES operations that require the persistency layer query for the same information regarding the monitored services. This information is then cached in memory by the database driver. Hence, after a few initial persistency layer queries, both the file-based and the database implementations use the memory for storing all the information required for SLAMES and there is no noticeable differences in their performance.

We should remember that the database implementation can be used by multiple SLAMES engine instances on different machines that share the same persistency layer. In that case, the performance of the database persistency will be affected by the remote connection to the database. The file-based implementation is useful for environments where the installation of a local database system is unwanted or expected to suffer from poor performance due to hardware limitations.
5.3.5 Performance of Java Code vs. Jython Scripts

Both policy conditions and actions can be written as Jython scripts. This section compares the performance between those scripts and their Java equivalent implementation in order to realize the expected performance price for using sophisticated Jython scripts within our policies.

Table 5.2 shows the execution time of a simple condition evaluated in a Java event-based condition against a Jython time-based condition. The Jython implementation is slower at a factor of about 1400 than the equivalent Java condition. As for actions, we can see that Jython script actions are slower than Java actions with a factor higher than 70.

In spite of the poor performance of Jython scripts, they are still encouraged for their powerful and flexible capabilities. Time-based conditions are evaluated only once in a given period. When choosing a timeout period in units of seconds rather than milliseconds, the few milliseconds it takes to evaluate the time based-condition is not significant for the overall performance of SLAMES.

Jython actions are slower in respect to Java actions, but we should remember than such actions are executed only upon violations of the policy. For most policies, these violations are expected to be rare as they usually deal with extreme scenarios that should not happen frequently. In case violations are likely to occur frequently, we encourage to use Java actions. For the rest of the cases, Jython actions are called rarely enough so that their poor performance is not critical for the operation of SLAMES.

5.4 Evaluation Summary

In this chapter we have evaluated different aspects of SLAMES performance. We compared the performance of different policies in SLAMES against each other and with respect to other monitored services.

In general, we conclude that the major part affecting SLAMES overhead is the operating system level network traffic and message processing time. This part is the result of SLAMES design decision to use the proxy approach. Other approaches may significantly reduce SLAMES overhead. We also conclude that local installation is preferred when the load of Web service calls is relatively low. A local database installation is recommended if such an installation is not expected to cause major performance degradation due to the hardware limitations of the machine on which SLAMES is installed. When writing policies, we see no significance in choosing Jython or Java actions unless one expects frequent violations. We also recommend to use time-based conditions with time interval higher than one second in order to ensure that Jython conditions are not evaluated at a high rate.
Chapter 6

Related Work

The subject of declaration and provisioning of non-functional requirements of Web services has been thoroughly studied in recent years. In this chapter, we refer to several works done in this area and point out differences between SLAMES and these studies.

6.1 SLAs for Web Services

Service Level Agreements for Web services have an essential role in the adoption of Web services in the business world. While the importance for Web services SLAs was well understood, it took several years for leading Web services framework vendors to agree on standard specifications in some aspects of SLAs. Other aspects remain with no standard, and are provided by the vendors with their own propriety implementation. Studies in this area, made both in academia and industry, lead to these accepted standards and suggest solutions for various issues. We can divide the studies in this area into two major types: studies that define SLAs for the purpose of service matching, and studies that provide continuous monitoring and enforcement of SLAs.

In Chapter 1, we mentioned the problem of service matching. When a service consumer wishes to use a service, it uses a lookup mechanism in order to find the services that provide the required functionality. Then, it has to choose the most appropriate service according to its non-functional capabilities. Service matching is the process of choosing the most appropriate service according to some heuristics or a pre-defined policy. For solving the service matching problem, most studies define their own ontology for expressing the non-functional requirements of Web services. The ontology of these studies is different from the policy used in this paper, since their purpose is to merge a policy representing the service provider capabilities and a policy representing the service consumer requirements.

Although service matching is not in the focus of this thesis, these studies are relevant as a reference for different approaches for expressing some quality-of-service constraints, specifically performance and availability oriented constraints. They are also useful for better understanding the requirements and capabilities needed in policies for SLAs. Since they focus on comparison and merge of policies, they tend to introduce rich expressive policy languages that better describe the business requirements of the service consumers. However, these languages are much more difficult to enforce and monitor.
Maximilien and Singh use a similar approach to SLAMES in their Web Services Agent Framework (WSAF) [22, 23]. In order to choose the most suitable service implementation for a consumer, the framework uses two-phase proxy mechanism. As in the integrated SLAMES topology, a proxy agent is installed on the service consumer, providing the same interface as the real service provider. However, their agent does not communicate with the real service implementation as in SLAMES, but interacts with a second service agent installed on their framework servers, which invokes the real service implementation. This service agent is responsible for monitoring the performance of the real implementation and spreading this information among other service agents using service agencies. It is also responsible for choosing the appropriate implementation by considering the consumer QoS preferences, the advertised service implementation QoS capabilities and the reputation of the service implementation, calculated in the service agencies according to the monitored data using their own trust model. Unlike SLAMES, their framework is meant for choosing an implementation and not for enforcement of the policy. Hence, WSAF framework does not try to solve SLA violations. It also monitors the performance from the WSAF server environment rather than the consumer environment, an approach that ignores QoS-related problems in the connection between the service consumer and the WSAF servers.

Zeng et al. invented the AgFlow system [38] on top of their Self-Serv framework [28] for provisioning of composite dynamic Web services [5]. Their framework focuses on composite Web services, which are Web services that depend on other Web services in their operation. In addition to choosing the appropriate service from several implementations, they apply a runtime provisioning mechanism that reacts to changes in the chosen services and re-initiates the service matching when service monitored data is changed. Like SLAMES, their QoS computation model (described in [18]) uses the consumer environment in order to monitor the performance of Web services, but later this information is sent to some QoS registry by the consumer itself. Unlike SLAMES, the framework is far from being transparent to both the consumer and the provider, demanding them to implement some extra functionality in order to work with the framework.

Another interesting framework is GluQoS [33]. Like SLAMES, its ontology is based on an extension to the WS-Policy standard. This framework uses mediators in both service provider and consumer in order to match requested and provided guarantees. However, the paper does not deal with active monitoring of the Web services performance.

A specification draft for performance was suggested at the Indiana University Extreme Lab, called WS-Performance [34]. This draft specifies a minimalistic model for asserting latency and throughput.

IBM proposed an extension to WSDL standard called Web Services Endpoint Language (WSEL) that supports declaration of QoS attributes to services. This extension was not widely-accepted.

Tosic et al. have proposed another language supporting the declaration of non-functional requirements called Web Services Offerings Language (WSOL) [30]. In their approach, a service may offer several packages of Quality-of-Service, called offerings, each one representing a set of non-functional requirements and a price. A service consumer should choose a preferred offering and in case of violations, choose a different offering from the same Web service to use, most likely to have weaker constraints on Quality-of-Service. If it fails to find other suitable offering, then it tries to find a different Web service providing the same
functionality in the known techniques. This approach for several offerings of the same service can be simulated in SLAMES using several policies. Upon violation of a given policy, the defined action can be programmed to use the management API in order to change the policy associated with this service.

WS-Agreement [3] is yet another specification dealing with SLAs for Web services. The specification defines the format and the terms for an agreement between the service provider and the service consumer. It also defines a protocol for making such an agreement by proposal of an agreement by one party and its approval by the other party. It is aimed for performance QoS guarantees, but does not specify who should monitor and enforce these constraints and associate an agreed penalty in case of SLA violation rather than actions to solve the problem, as proposed by SLAMES. An extension to this specification was suggested by Modica et al. in [7] in order to improve handling with violations. They propose re-negotiation of the involved parties in case of SLA violations in order to agree on a new modified agreement that both parties are satisfied with. SLAMES can be used to enforce an equivalent behavior using the management API as specified earlier.

Probably the most comprehensive study on declaration and continuous monitoring of SLAs was carried out by A. Keller and H. Ludwig. Their WSLA framework [15] defines an ontology for representing SLA policies [19] including constraints for performance and availability, and provides a high-level architecture design of a system that monitors and enforces the policies. The WSLA framework does not specifically define how performance is measured by its measurement service. It leaves the choice whether to probe this information from inside the provider or from the consumer environment to the specific implementation. Since the framework does not define this issue, it is hard to tell how transparent is the framework to the parties. The WSLA framework also defines corrective actions to SLA violations. These actions are intended for reporting about violations, but can also be used to propose a solution to the service provider. The intent of the authors was that these actions should be done by the service provider and not the service consumer. Since it is not clear in which party the WSLA framework is installed, it is hard to tell whether this framework has SLAME-like capabilities in solving network problems.

6.2 Network Monitoring

Provisioning of performance in computer networks has been studied long before Web services were invented. Measurement and performance statistics were collected and a common protocol - Simple Network Management Protocol (SNMP) [13] was used to advertise and retrieve the collected information.

Since Service Level Agreements are used by Web service consumers and providers, they use high-level definitions for monitored data, which is usually not the real information being physically monitored. Hyo-Jin Lee et al. created a mapping from QoS parameters guaranteed in some Web service SLAs to their relevant low-level monitored values, called Network Performance Metrics (NPMs) [17]. This mapping is relevant for implementation of a Web services provisioning system such as SLAMES, where monitored performance variables should be translated to the QoS variables defined in the SLA policy. This mapping is very useful if we want to extend SLAMES to provide more QoS variables.
6.3 QoS Issues In Other Frameworks

Web services frameworks are not the first attempt to provide an infrastructure for services over the network. Common Object Request Broker Architecture (CORBA) [12] was one of the first frameworks to support inter-operable execution of remote services. Hence, Quality-of-service topics are as relevant to CORBA as they are relevant for Web services.

Indeed, there are many ideas common to both CORBA and SOAP-based Web services as frameworks for publishing and running inter-operable services. Both use an interface language (IDL and WSDL) that is independent of the implementation language. Both provide some serialization protocols for marshalling operations into messages and vice versa.

However, CORBA provides the concept of interceptors while SOAP-based Web services lack this kind of feature. Specific implementations, such as Apache Axis or .Net Web services framework, support such mechanisms using their own propriety configurations. However, this is not part of the Web services standard.

CORBA Interceptors are built inside the framework. A detailed description of how to use this mechanism for implementing fault-tolerance, load-balancing and other QoS issues in CORBA is specified in [10]. Mediation and interception techniques for Web services are not standardized, since this feature violates the principles of the Web service model. The model defines a language for declaring an interface and a protocol for invoking operations, including serialization issues. However, the idea behind this model is to avoid forcing any local architecture. Since interceptor mechanism should be implemented on the local machine, such mechanism contradicts the philosophy of the model. However, the Web services model make an extensive use in the HTTP protocol, which makes it possible to use known HTTP-based mediation services, such as web proxies in order to achieve a similar behavior to CORBA interceptors.
Chapter 7

Summary and Future Directions

In this chapter, we summarize the results of the research work done in this thesis and conclude some insights regarding the monitoring and the enforcement of performance QoS in Web services. Then, we list some key directions that can lead to a further research in this area.

7.1 Summary

In this thesis we presented a lightweight system for monitoring the performance of Web services called SLAMES. The purpose of building SLAMES was to prove some key concepts in the world of Web service SLA enforcement in dynamic networks.

First, we claimed that monitoring in these networks should be done on the consumer environment due to the nature of these networks, where network instability is an important factor for the poor performance of these services. Since a monitoring system installed on the consumer environment encounters the same network problems as the service consumer itself, it is the ideal location of such system for the purpose of performance monitoring.

Installing a monitoring system on the service consumer is not suitable for all customers. While service providers ensure that their servers are strong enough to deliver a certain level of performance, service consumers are usually end-users whose machines are not necessarily strong. This is the reason for building SLAMES on top of existing lightweight frameworks. SLAMES was built to provide a relatively small overhead on the performance of Web services. In order to improve performance, SLAMES uses HTTP monitoring techniques that do not require to parse every SOAP message transferred between the parties, which is considered to be a major reason for the relatively poor performance of Web services with respect to other remote service models. Our measurements show that under some scenarios, the latency overhead of the entire SLAMES system is about half of the duration it takes to parse the SOAP messages in the protocol. However, this achievement comes at the cost of the flexibility of policy association. SLAMES can associate a policy only with an entire service rather than specific operations delivered by a service.

Another key concept involved in choosing SLAMES to operate in the consumer environment is the ability to solve some of the network problems by the means accessible to the service consumer. These means include a network path discovery renewal of the path to the
service provider or alternatively finding a network path to a different mirror of the service, which is possible only with some level of support from the service provider. To the best of our knowledge, a consumer-side approach for solving poor performance of Web services was not used before.

The flexibility of the system is also a key feature of SLAMES. One level of flexibility is reached by using a standalone proxy approach for SLAMES. This approach enables the service consumer to use SLAMES regardless of the Web services framework or platform used by the consumer, as long as there is a JVM (Java Virtual Machine) implementation for that platform. Another level of flexibility is reached within the policy itself. By giving the policy writer the power of a scripting language, there is much higher chance that SLAMES can be customized to provide solution to the problems encountered in a specific environment. Scripting languages are known for their capabilities to provide quick solutions in such scenarios.

To sum it up, SLAMES is a monitoring system providing solutions to poor Web services performance with a focus on problems that occur especially in dynamic networks with a relatively low overhead and flexible capabilities. SLAMES policies are different from standard policies since they also define the behavior of the system upon SLA violations, yet they can be integrated with existing standard policies in a WS-Policy based policy.

7.2 Future Research Directions

In chapter 4, we discussed the proxy approach used by the SLAMES engine in order to enable the monitoring and the enforcement of the policies. We tried to reduce the overhead of this approach by the techniques mentioned earlier, but a minimal overhead will always remain because of this approach. Further research may be conducted on applying a network traffic monitoring approach when implementing the core functionality of SLAMES engine. This approach has more limitations on the network topology between the service consumer and SLAMES installation machine, probably demanding that SLAMES should always be installed on the same machine as the service consumer. However, under good network performance circumstances where there is no violation in the policy, this approach is expected to have a zero overhead on the performance of the Web service.

Another direction that is yet to be studied is the cooperation between SLAMES and other Web services monitoring frameworks installed on the service provider environment. Such cooperation may lead to more accurate diagnosis of the encountered network problems and hence, an option to employ the appropriate solution to the problem, which may involve actions done by both SLAMES and the service provider framework.

This kind of cooperation is also expected to improve the availability detection of SLAMES. Since SLAMES monitors the performance in a passive manner, it cannot calculate the availability rate of a service if it was not accessed recently. A more active approach is to send a ping-like message to the service provider if it was not used for a specified period of time. The service provider is required in such case to support an operation that provides an indication about its availability status. This approach was used in the implementation of Fault Tolerant CORBA framework [26]. It is possible to implement this active approach in the current implementation of SLAMES using time-based conditions that execute Jython scripts. Extending SLAMES to use active monitoring techniques requires a study to better
understand the implications of active monitoring techniques on the performance of dynamic networks.

As stated in the previous section, SLAMES low overhead is gained at the cost of the possibility to apply a policy to a specific operation in a Web service. Further work on this matter can lead to eliminating this lack of feature. A possible approach, inspired by the REST Web services model, is to split operations with different policies to different proxy addresses in order to identify the operation in the URL of the proxy. Different approaches may be to try to extract the information about the Web services operation from HTTP headers or by a quick limited parsing of the SOAP message itself.

Finally, additional study can be done in order to share the knowledge about network problems between monitored services. It is very likely that a poor network performance affects more than one monitored service in SLAMES. Dealing with violations of all these services at the same time may cause redundant operations to be executed in the optimistic scenario or prevent the success of such solving actions in the worst scenario. The implications of solving a specific monitored service performance problem on other monitored services on the same machine should also be addressed in a further study.
Appendix A

Slames Policy Schema Definition

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema
targetNamespace="http://www.cs.technion.ac.il/ws/slames/1.0/policy"
elementFormDefault="qualified"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://www.cs.technion.ac.il/ws/slames/1.0/policy">

<complexType name="QosPolicyType">
    <sequence>
        <element name="configuration" type="tns:ConfigurationType" maxOccurs="1" minOccurs="0"></element>
        <element name="event" type="tns:EventType" maxOccurs="unbounded" minOccurs="1"></element>
    </sequence>
</complexType>

<complexType name="EventType" mixed="true">
    <sequence>
        <choice>
            <element name="condition" type="tns:EventConditionType" />
            <element name="timed-condition" type="tns:TimedConditionType" />
        </choice>
        <element name="action" type="tns:AvailabilityActionType" />
    </sequence>
</complexType>

<complexType name="TimedConditionType" mixed="true">
    <attribute name="interval" type="positiveInteger" use="required"></attribute>
    <attribute name="value" type="string" use="required"></attribute>
</complexType>
```

47
<attribute name="type" use="required">
  <simpleType>
    <restriction base="string">
      <enumeration value="jython" />
    </restriction>
  </simpleType>
</attribute>
</complexType>

<complexType name="EventConditionType">
  <choice>
    <element name="latency" type="tns:LatencyType" />
    <element name="throughput" type="tns:ThroughputType" />
    <element name="averageLatency" type="tns:AverageLatencyType" />
    <element name="load" type="tns:LoadType" />
    <element name="availability" type="tns:AvailabilityType" />
  </choice>
</complexType>

<complexType name="AvailabilityActionType" mixed="true">
  <sequence>
    <element name="type" type="tns:ActionTypeType" minOccurs="1" maxOccurs="1" />
    <element name="file" type="string" minOccurs="0" />
  </sequence>
</complexType>

<simpleType name="ActionTypeType">
  <restriction base="string">
    <enumeration value="reroute" />
    <enumeration value="print" />
    <enumeration value="jython" />
  </restriction>
</simpleType>

<complexType name="LatencyType">
  <attribute name="minMillis" type="nonNegativeInteger" />
  <attribute name="maxMillis" type="positiveInteger" />
</complexType>

<complexType name="ThroughputType">
  <attribute name="minOperationRate" type="tns:nonNegativeDouble" />
  <attribute name="maxOperationRate" type="tns:positiveDouble" />
</complexType>
<complexType name="AverageLatencyType">
    <attribute name="minMillis"
        type="nonNegativeInteger"/>
    <attribute name="maxMillis"
        type="positiveInteger"/>
</complexType>

<complexType name="nonNegativeDouble">
    <restriction base="double">
        <minInclusive value="0" />
    </restriction>
</complexType>

<complexType name="positiveDouble">
    <restriction base="double">
        <minExclusive value="0" />
    </restriction>
</complexType>

<complexType name="ConfigurationType">
    <choice>
        <element name="latency-history-millis"
            type="positiveInteger"/>
        <element name="load-history-size"
            type="tns:TimeSizeAndUnitType"/>
        <element name="throughput-history-size"
            type="tns:TimeSizeAndUnitType"/>
        <element name="availability-period-minutes"
            type="positiveInteger"/>
        <element name="no-violation-timeout-minutes"
            type="nonNegativeInteger"/>
    </choice>
</complexType>

<complexType name="TimeSizeAndUnitType">
    <restriction base="string">
        <pattern value="([0-9])+(s|m|h|d)" />
    </restriction>
</complexType>

<element name="policy" type="tns:QosPolicyType"/>

<complexType name="LoadType">
    <attribute name="minRequestRate"
        type="tns:nonNegativeDouble"/>
    <attribute name="maxRequestRate"
        type="tns:positiveDouble"/>
</complexType>
<complexType name="AvailabilityType">
    <attribute name="min" type="tns:PercentType"></attribute>
    <attribute name="max" type="tns:PercentType"></attribute>
</complexType>

<simpleType name="PercentType">
    <restriction base="float">
        <minInclusive value="0.0"></minInclusive>
        <maxInclusive value="1.0"></maxInclusive>
    </restriction>
</simpleType>
Appendix B

Evaluated Policy Definitions

B.1 Throughput-based Policy

```xml
<wsp:Policy
 xmlns:slm="http://cs.technion.ac.il/ws/slames/1.0/policy"
 <slm:policy>
   <slm:event>
     <slm:condition>
       <slm:throughput maxOperationRate="4000"/>
     </slm:condition>
     <slm:action type="print">There is violation</slm:action>
   </slm:event>
 </slm:policy>
</wsp:Policy>
```

B.2 Latency-based Policy

```xml
<wsp:Policy
 xmlns:slm="http://cs.technion.ac.il/ws/slames/1.0/policy"
 <slm:policy>
   <slm:event>
     <slm:condition>
       <slm:latency minMillis="3000" maxMillis="60000"/>
     </slm:condition>
     <slm:action type="reroute"/>
   </slm:event>
   <slm:event>
     <slm:condition>
       <slm:latency minMillis="60000" />
     </slm:condition>
     <slm:action type="jython">print "Response Time is %d milliseconds" % latency</slm:action>
   </slm:event>
 </slm:policy>
</wsp:Policy>
```
B.3 Intensive Policy

```xml
<wsp:Policy
  xmlns:slm="http://cs.technion.ac.il/ws/slames/1.0/policy"
  <slm:policy>
    <slm:event>
      <slm:condition>
        <slm:latency minMillis="3000" maxMillis="300000"/>
      </slm:condition>
      <slm:action type="jython">
        print "Response Time is " + averageLatency + " milliseconds"
      </slm:action>
    </slm:event>
    <slm:event>
      <slm:condition>
        <slm:load minRequestRate="200.1" maxRequestRate="6000.5"/
        <slm:throughput minOperationRate="150.3" maxOperationRate="5000.5"/>
        <slm:averageLatency minMillis="3000" maxMillis="600000"/>
        <slm:availability min="0.0" max="0.4"/>
      </slm:condition>
      <slm:action type="reroute"/>
    </slm:event>
    <slm:event>
      <slm:timed-condition interval="2000">(averageLatency &gt; 300000) and (availability &lt; 0.9)</slm:timed-condition>
      <slm:action type="print">Timed Condition Violation</slm:action>
    </slm:event>
  </slm:policy>
</wsp:Policy>
```
References


[31] W3C. Extensible Markup Language. URL: http://www.w3.org/XML.


אכיפת איזומות עבורי שירוטי רשת ברשות דינהית

ניר שמי
אכיפת آıcכח עבככ שירורית יש ברשחית דיןימית

תיער על מחקור

ליש מיילך חלך על הדרישות לקבעת חוויה
מנסים למעיע מִדִּיע מִמֵּחֲשָׁב

גֵּי שְׁמִי

חגש לכססי התכשיטי — מכון טכנולוגי לישראל
חיפה
מרץ 2009

אֶדְרַהֶה הֶשָּׁפֶל
המחקך ענשיו בורחיות פרופ' תמר רעי פרידמן מをつけעה לימודים מחשב

בראינו הרדוע תליון פרידמן על הכותב שמשלח בחרק תחת הינוית במקליע גבות אופן המחקך.
נכתבו埃尔 הדוקוטו, ענשהו והמטימה רכ אל שראוור ואצ החתמח המחקך.
בראינו הרדועת ליפור, שרה ומכה ימי על הורמיה והתידדה לכל אורח הנקבות המחקך.

אני פהדה ועלים חכמי העתים הכסיית אורודיה בחשדלות

טכניון מחקר

בחוקה: בחלקים האיגוריים והגאומטריים, ימחק את הקטגוריה הפוסט-בלוק במפגש, ויביס טכניקת התמחות.

המתקדמים וنشاطם בпечатת תכנית - בה גם ברקע של יישום תכנית, והدافע בחברות
ולא בפרטים אינפרא-דווריים, שיתוף פעולה במדיום של ייצוג, במג [][]

הסとはいえ, בסופו של דבר, יSweden המגמה במדיום של ייצוג, במג [][]

אולקטרה

על בטח לסקפ או ברifiable משקפת, ובעית התחタイミング אצולה של יישומי
ולא יישומי מיילור, במגמה במדיום של ייצוג, במג [][]

ה אותך והتجار בשירות, SOA והаютוע עריכת מוסדות של שיתוף פעולה
ובנוסף שיתוף הפעולה בשירות, SOA והאתוע עריכת מוסדות של שיתוף פעולה

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

טכניון מחקר

WS-Policy describes the way in which a service exchange is performed, how the service is described, and how the service is accessed.

XML Policy, which is based on XML, is used to describe the service. XML Policy is a standard for describing services and is widely used in the industry.

WSDL (Web Services Description Language) is another standard for describing services. WSDL provides a formal notation for describing the service, including the service's operations, inputs, outputs, and bindings.

UDDI (Universal Description, Discovery, and Integration) is used to discover services. UDDI is a registry of services, where services can be registered and discovered by other services.

WS-PolicyAttachment is a standard for describing the attachment of a service to a policy. This is useful for describing how a service is related to a policy, such as how a service is described in a policy.

Web Services Offerings Language (WSOL) is a standard for describing the offerings of a service. WSOL is used to describe the features and capabilities of a service, including the service's operations, inputs, outputs, and bindings.

Web-service Service Level Agreement (SLAMES) is a standard for describing the service level agreement of a service. SLAMES is used to describe the obligations of a service, including the service's performance, reliability, and availability.

DSR (Distributed System Registries) is a standard for describing the distributed system registry of a service. DSR is used to describe the registry of a service, including the registry's location, name, and description.

SLACES (Service Level Agreement for Clouds) is a standard for describing the service level agreement of a cloud service. SLACES is used to describe the obligations of a cloud service, including the service's performance, reliability, and availability.

The Web Services framework is a set of standards and specifications that define how services can be described, discovered, and accessed. The framework is based on XML and includes standards for describing services, discovering services, and accessing services.

The framework includes standards for describing services, such as WSDL and XML Policy, and standards for discovering services, such as UDDI. The framework also includes standards for accessing services, such as SOAP (Simple Object Access Protocol) and REST (Representational State Transfer).

The framework is used to describe services, such as web services and cloud services, and to discover services, such as services that are available on the internet. The framework is also used to access services, such as services that are available on the internet or on a local network.
The device, known as a computer, is a machine designed to process information. It consists of hardware and software components. The hardware includes the physical devices such as the central processing unit (CPU), memory (RAM), storage devices (HDD, SSD), input devices (keyboard, mouse), and output devices (monitor, printer). The software includes the operating system (OS), applications, and system utilities.

The OS acts as a intermediary between the hardware and the user. It manages the resources of the computer and provides an interface for users to interact with the system. It also manages various tasks and processes running on the computer, ensuring that they run efficiently and do not interfere with each other.

Applications are software programs designed to perform specific tasks. They can range from simple utilities like notepad and calculator to complex programs like word processors and video editing tools. Applications require the system to allocate resources such as memory and processing power, which the OS manages to ensure smooth operation.

System utilities are programs provided by the operating system to perform maintenance and diagnostic tasks. These include tasks such as creating backups, managing files, and troubleshooting hardware issues.

In summary, the computer is a sophisticated device with complex hardware and software components. The OS acts as a coordinating element, managing resources and providing a user-friendly interface. Applications and utilities are the tools that allow users to perform specific tasks and manage their devices.