Ensuring Service Level Agreements for Composite Services by Means of Request Scheduling

A Thesis Submitted to the

College of Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the degree of Doctor of Philosophy

in the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Dmytro Dyachuk

Permission to Use

Abstract

Building distributed systems according to the Service-Oriented Architecture (SOA) allows simplifying the integration process, reducing development costs and increasing scalability, interoperability and openness. SOA endorses the reusability of existing services and aggregating them into new service layers for future recycling. At the same time, the complexity of large service-oriented systems negatively reflects on their behavior in terms of the exhibited Quality of Service. To address this problem this thesis focuses on using request scheduling for meeting Service Level Agreements (SLAs). The special focus is given to composite services specified by means of workflow languages.

The proposed solution suggests using two level scheduling: global and local. The global policies assign the response time requirements for component service invocations. The local scheduling policies are responsible for performing request scheduling in order to meet these requirements. The proposed scheduling approach can be deployed without altering the code of the scheduled services, does not require a central point of control and is platform independent.

The experiments, conducted using a simulation, were used to study the effectiveness and the feasibility of the proposed scheduling schemes in respect to various deployment requirements. The validity of the simulation was confirmed by comparing its results to the results obtained in experiments with a real-world service. The proposed approach was shown to work well under different traffic conditions and with different types of SLAs.

Acknowledgements

I am deeply grateful to the following people for their help and support during the period I have spent at the Department of Computing Science.

I would like to express my sincere gratitude to my supervisor, Dr. Ralph Deters, for his support, guidance and insight as well as for the many valuable suggestions he made. His continuous financial support and understanding were especially helpful in the final years of the program.

Also I would like to thank my advisory committee: Dr. Winfried Grassmann, Dr. Derek Eager, Dr. Mark Keil and Dr. Raj Srinivasan. Your advice and comments always were very helpful.

Thanks to the people that have been around me during this long journey. A special thank you goes to Dr. Michelle Mazzucco for being a good friend, research partner and sharing some of the source code which was used in the simulations. Also, I would say thank you to Valery Kotlyarova for being a good supportive friend and an excellent thesis writing companion, Shayne Feairs for being a great friend and helping me with the adaptation.

I am greatly grateful to my parents Oleg and Tetyana for supporting and encouraging me. Without their constant support I would not have made it.

Finally, I would like to thank Jan Thompson for providing excellent administrative support.

Contents

List of Tables

List of Figures

List of Symbols

Symbol	Meaning	Definition
ϕ	admission rate used by GPS	-
Ω	workflow segment	-
t	current time	-
r_j	job j arrival time or release time	-
p_j	job j processing time or job size	-
C_j	job j completion time	-
D_j	job j relative deadline	-
d_j	job j absolute deadline	$d_j = D_j + r_j$
$\hat{D}_j$	job j relative hard deadline	-
$\hat{d}_j$	job j absolute hard deadline	$\hat{d}_j = \hat{D}_j + r_j$
y_j, s_j	job j laxity (slack time)	$s_j = y_j = d_j - (t + p_j)$
a_j	job$j$ allowance	$a_j = d_j - t$
L_j	job j lateness	$L_j = C_j - d_j$
E_j	job j earliness	$E_j = max(0, d_j - C_j)$
T_j	job j tardiness	$T_j = max(0, C_j - d_j)$
w_j	job j weight	-
$V_j(t)$	revenue for completing job j at time t	-
$P_j(t)$	penalty for rejecting(aborting) job j at time t	-
v_j	revenue for completing job j before deadline d	-
p_j	penalty for completing job j after deadline d_j but before $\hat{d}_j$	-
$\hat{p_j}$	penalty for violating hard deadline $\hat{d}_j$	-
W_j	job j waiting time	$W_j = C_j - r_j - p_j$
F_j	job j flow time or response time	$F_j = C_j - r_j$
U_j	job j unit penalty	$U_j = \begin{cases} 0 & C_j < d_j \\ 1 & C_j \geq d_j \end{cases}$

List of Acronyms

A/OPN	Allowance Per Operation
BPEL4WS	Business Process Execution Language for Web Services
C-SFQ(D)	Weighted Fair Queueing
CPM	Critical Path Method
DASA	Dependent Activity Scheduling Algorithm
EF	Equal Fraction
ESB	Enterpirse Service Bus
EDF	Earliest Deadline First
EQF	Equal Flexibility
GPS	Generalized Processor Sharing
GUS	Generic Utility Scheduling Algorithm
HRF	Highest Ratio First
HVF	Highest Value First
LBESA	Lockes Best Effort Scheduling Algorithm
LSM	Lawlers Scheduling Method
MLP	Multi Level Policy
MPL	Multiprogramming Level
MST	Minimal Slack Time
OPTCON	Decomposing into Linear-Constant TUF
PF	Proportional Fraction
PGPS	Packet GPS
PI	Proportional Integral
PQF	Proportional Flexibility
PP-aware	Penalty Revenue Aware scheduling
PUD	Potential Utility Density
QoS	Quality of Service
RMS	Rate Monotonic Scheduling
S/OPN	Slack Per Operation
SCALL	Scaling Based in TUF shape
SCEQF	Scaling based on EQF
SCR	Smallest Critical Ratio
SCV	Squared Coefficient of Variation
SFQ	Start-Time Fair Queueing
SJF	Shortest Job First
SLA	Service Level Agreement
SLO	Service Level Obligations
SOA	Service-Oriented Architecture
TUF	Time/Utility Function
UD	Ultimate Deadline
UT	Ultimate Time/Utility Function
WCF	Windows Communication Framework
WF^2Q	Worst-Case Weighted Fair Queueing
WFQ	Weighted Fair Queueing

Chapter 1
Introduction

This chapter introduces the main concepts used throughout the thesis and it presents the motivation for the work, the scope of research, as well as the proposed solution.

1.1 Basic Concepts

The following section contains a description of the basic concepts required for understanding the context of the problem addressed in the thesis. The notions of Service Oriented Architecture, Quality of Service, Service Level Agreements, composite services, service workflows, worklflow patterns and Enterprise Service Bus are defined and explained.

1.1.1 Service Oriented Architecture

The Service-Oriented Architecture (SOA) was first introduced in 1996 by the Gartner Group [117]. SOA refers to principles used in engineering large distributed systems, rather than to a particular communication/middleware technology. McCoy and Natis [98] define SOA in the following way: "Essentially, SOA is a software architecture that builds a topology of interfaces, interface implementations and interface calls. SOA is a relationship of services and service consumers, both software modules large enough to represent a complete business function". More specifically, SOA envisions distributed systems as collections of services harnessed together. Each service represents a unit of business logic, presumably a business process. Box [23] defines the four primary tenets of service orientation:

- Tenet 1 (boundaries are explicit): Each service has a contract associated with it. The contract is the only "gate" of communicating with a service.

- Tenet 2 (services are autonomous): Services communicating with other services should take into consideration possible network and services failures.
- Tenet 3 (services share schemas and contracts, not classes): Classes encapsulate the behaviour of a class as well as its attributes, while schemas and contracts contain only description of the data format.
- Tenet 4 (service compatibility is based on policy): Each service has a set of policies reflecting the properties of interaction with a service, e.g. security protocols, transactional properties.

Currently, systems designed according to SOA are based on Web Services. Although SOA does not demand using a particular technology [98], Web Services allow all four tenets of service orientation and are most widely supported by a vast range of software vendors.

1.1.2 Quality of Service

According to ISO 8402 [67], the quality of software is defined as "The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs". However, those needs can be interpreted in various ways depending on a domain of an application. Researchers [96, 114, 76] define Quality of Service (QoS) for technologies used in SOA. Although definitions of the basic QoS metrics in literature vary somewhat, one of the most common ways of expressing QoS in a measurable way is the following:

- Availability: Availability [96] describes the ratio of the time the service is available for accepting requests over the total time. Essentially, it represents the fraction of the time a service is ready for offering correct service [96].
- Performance: Performance includes such metrics as latency, service response time, service throughput and transaction time [96].
  - Throughput is defined as the maximum number of requests served per unit of time.
  - Response time refers to the time required for processing a single request.

  - Latency is the time elapsed since a request from a client has been sent and a corresponding response was obtained. From a client's perspective, latency also includes network latency, i.e. time needed for delivering a message.
  - Transaction time refers to the amount of time required for completing a transaction, which can comprise several service calls.

- Reliability: Reliability refers to the ratio of the successful service invocations over the total number of service calls: $Reliability = \frac{SuccessfulServiceInvocations}{AllServiceInvocations}$. A service invocation is considered successful if it does not result in an exception or failure on the service site and if the response is returned in a correct time frame, usually defined by a transport protocol [96].

In this thesis, the focus is given to the QoS metrics describing the performance of a service, more precisely to its response time.

1.1.3 Service Level Agreements

Agility is one of the key intentions driving the migration of large distributed applications towards SOA. In the "modern" business world, customization of applications often determines the success of a particular business. Hence, being able to adapt to rapid market changes significantly increases the competitiveness of the company. SOA achieves this agility by means of using service composition. Industrial experience of implementing SOA showed that combining business processes aligned as services significantly reduces the time for delivering new solutions based on the existing infrastructures[1]. Using third-party services became a common practice in most companies, e.g most of the online shops use credit card services, shipping services, etc.

Nevertheless, seamless integration of outsourced services entails a complete dependence on non-functional aspects of incorporated components. Fluctuations in their dependability or performance negatively reflect on the stability of the incorporating systems. For example, the unexpected slowdowns in employed third-party services can slow down the applications

[1]http://www-306.ibm.com/software/integration/wmqwf/editions/

in the company itself. Naturally, in order to demarcate the responsibilities, it is necessary to establish a contract between service consumers and service providers that defines the functional and non-functional requirements. The latter should also include a description of the obligations from a service side in terms of performance, availability, reliability etc. Such a contract is known as a Service Level Agreement (SLA). Each SLA contract defines "assertions of a service provider to perform service according to agreed guarantees for IT-level and business process level service parameters" [75]. Usually a contract includes signatory parties or contractual parties (service providers and service consumers) and a third party (monitoring) as well as a service description that contains possible bindings for service operations, transport protocols and SLA parameters. It is common practice for contracts to be customized for each service consumer. A study of over 300 SLAs [75] shows that even similar SLAs have highly diverse semantics and even simple parameters, e.g. throughput, can be defined in multiple ways. Thus, SLAs have the ability of expressing customized metrics through basic metrics (e.g. response time, downtime), mathematical expressions and parametrized statistical functions (e.g. mean). The restrictions imposed on a service as well as on its specific methods result in Service Level Obligations (SLO), e.g. maximum response time is less than or equal to a certain value. Moreover, SLA should also stipulate a measuring directive – a method of obtaining monitored parameters, e.g. directly from a service or by means of a proxy. The measuring directive is often coupled with the precise specification of measurement intervals, their duration and frequency. The last key parameter describes the billing process, e.g. a price associated with each successful service invocation and a penalty in case of an SLA violation. An adoption and an augmentation of SLA for SOA resulted in the emergence of multiple standards allowing precise agreements definitions, e.g. WSLA [75] [38], WSOL [127] [128], SLAng [80] QML [57], WSML [115] and WS-Agreement [6].

1.1.4 Composite Services and Service Workflows

Service composition implies combining multiple services into a new service, also known as a composite service. Composite services are offered for client consumption, or can serve as building blocks for future compositions [109]. Composite services are responsible for coordinating the work of composition participants, managing their control and data flow.

Workflow languages can be viewed as 4G languages [99], as they appear to be highly platform independent languages for describing composite business processes. However, at the same time workflow languages are very expressive, e.g. BPEL4WS is Turing complete [51].

Defining composite services by means of workflow languages is one of the most common practices in SOA. Adopting workflow languages for specifying composite services provides a set of advantages over solutions employing canonical programming languages, such as Java, C# and others. Those advantages include easier development, monitoring and management. Workflows languages, originated from the business management community, are supported by a large number of process visualization and editing tools. These tools allow assembling services in workflows without understanding of technical details, such as networking stack, threading model, etc. The fact that they are domain specific languages providing a natural and simple way of creating long running transactions, compensation events and event management is another key argument in the favour of using workflow languages. After all, most of the workflow execution engines[2] offer full management and monitoring of the workflow execution.

On the other hand, specifying service orchestrations using standard 3G languages (Java, C#, etc) enables a wider spectrum of possibilities, such as code optimization and do not require developers learning another programming language. However, these advantages usually come at a price of higher development costs, as they require involvement of more professional developers.

Note that in the remaining part of the thesis, workflows and composite services will be used as interchangeable terms, as it is being assumed that composite services are being defined using a workflow language such as the Business Process Execution Language for Web Services (BPEL4WS) [5].

Van der Alst et al. analyzed various types of workflows used in practice [132] and identified the main patterns used for performing service aggregation. The basic patterns include sequence [Figure 1.1], parallel split & synchronization [Figure 1.2], exclusive choice [Figure 1.3] and arbitrary cycles [Figure 1.4]. More advanced patterns include static partial join for multiple instances, multiple-choice, multi-merge, structured discriminator and others. This

[2]for example ActiveBPEL – http://ActiveVOS.com/BPEL, Apache ODE – http://ode.apache.org/

Table 1.1: Types of workflows patterns with respect to the determinism of their behaviour.

Static	sequence, parallel split
Dynamic	exclusive choice, loop

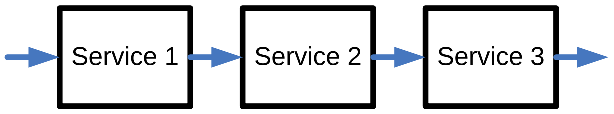

Figure 1.1: Sequence pattern.

thesis focuses solely on the worklfows composed using the basic patterns which are defined in the following paragraph.

Sequence pattern refers to the situation when each service in the chain is invoked only after the response from the previous service has been received [Figure 1.1]. In the case of the parallel split pattern (also known as split), all components are invoked at once [Figure 1.2]. Exclusive choice is used in the situations when only one of the services is invoked. Note that exclusive choice is an invocation pattern with non-deterministic behaviour, as the decision, regarding which branch will be invoked, is performed at runtime. Another pattern with non-deterministic behaviour is arbitrary cycles (also known as a loop) [Figure 1.4]. Loop represents an unknown number of repetitions of the specified service invocation. Patterns with non-deterministic behaviour from now on in this thesis will be regarded as dynamic patterns and the ones with deterministic behaviour as static patterns. Consequently, workflows consisting strictly of static components will be called static services or workflows, while the ones having at least one dynamic component will be called dynamic workflows [Table 1.1]. Note that patterns can used recursively, i.e. there could a sequence of loops, a loop of splits, etc.

1.1.5 Enterprise Service Bus

The Enterprise Service Bus (ESB), implementing a transit layer between services, is an inherent part of many complex service-oriented systems. Essentially, the ESB represents an additional layer [Figure 1.6], shielding the services from direct invocations [Figure 1.5].

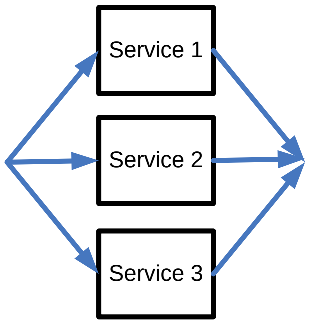

Figure 1.2: Parallel split and synchronization pattern(parallel jobs).

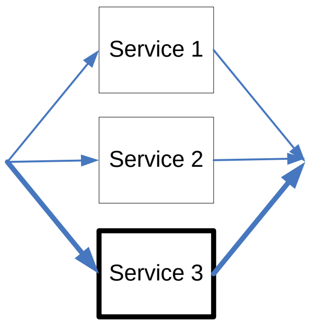

Figure 1.3: Exclusive choice pattern.

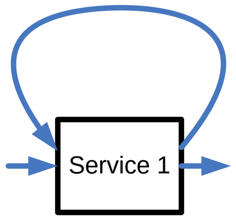

Figure 1.4: Arbitrary cycles pattern.

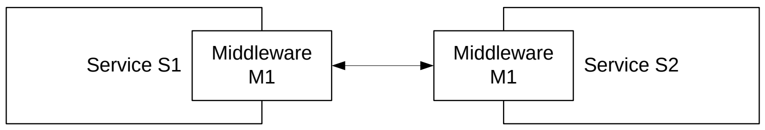

Figure 1.5: Direct service invocation.

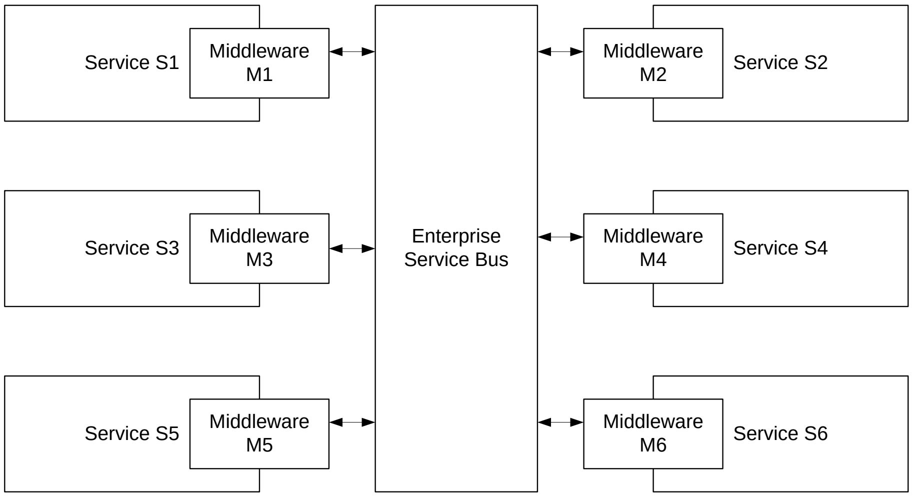

Figure 1.6: Service invocations using an Enterprise Service Bus.

The main advantages of using an ESB include [109]:

- Location transparency: In the classical direct style invocation pattern [Figure 1.5] service clients need to know the exact location of a service in order to invoke it. This creates a number of complications when, for example, a service goes down or changes its location. Employing an ESB allows shifting the burdening task of discovering service from the service clients' site to the ESB.

- Distributed messaging: Providing reliable and distributed transport for guaranteed delivery of messages to services in the presence of network failures is another problem in the direct point-to-point service invocation model. An ESB can implement a store-and-forward technique in order to make sure that a message will not be lost even if its transmission is disrupted by a network failure [108, 52].

- Message transformation: It is a common practice that formats of messages exchanged even within the same organization are mismatched. The objective of an ESB, in this case, consists of translating those different formats in order to enable simpler inter-operation between services and clients.

- Multi-protocol support: Depending on a service, the protocol used for transporting messages can range from as simple as TCP to as complex as the Advance Message Queueing Protocol [134, 105]. Consuming such services for one customer requires substantial additional effort in order to fully support all these protocols and thus, often results in high development costs. An ESB, on the other hand, enables mitigating this problem by performing the protocol transformation itself.

- Message routing: By means of routing incoming requests an ESB can implement load balancing in order to achieve higher scalability, implement fail-over for improved reliability, etc.

It is important to note that the ESB is usually hosted on a standalone dedicated server or even a set of servers.

1.2 Problem Definition

The main objective of this thesis is to *examine whether scheduling service requests can serve as a means for enforcing established Service Level Agreements stipulating response time guarantees for service workflows.*

1.2.1 Model

The more precise definition requires introducing a model first.

Each atomic service is hosted on a dedicated server. Request processing by an atomic service does not result in sending requests to any other services, nor does it impose any load on the hardware other than its own (see Tenet #1 from subsection 1.1.1). Each service can process a limited number of requests at the same time, if an arriving request sees that this number has reached its maximum it is parked in a queue. Once the request is placed in the queue it can be either rejected (due to a timeout) or accepted in service. Services do not support preemption, thus once the request has been admitted in service it only can be processed. Job sizes are distributed with parameters known ahead of time and can be predicted.

A composite service request triggers invoking component services and combining their responses in a response returned to the client. The dependencies between those invocations are defined using the following patterns: the sequence, the parallel split, the exclusive choice and the loop (described in Section 1.1.4). Patterns can be recursive. For example, a sequence can be a sequence of loops, where each loop contains a parallel split. The dependencies between service invocations are specified by means of a workflow.

The precedence relations between component service invocations in the workflow designed using the exclusive choice and the loop patterns are not deterministic, i.e. the selection of a specific branch is not known until runtime, neither is the number of loop iterations. However, in the case of the exclusive choice, the probabilities of the branch selections are known and in the case of the loop the probability density of the number of iterations is available.

An atomic service can be a part of several compositions at the same time. Atomic services are exposed only to the load generated from composite services. In the case where an atomic service is being invoked by clients directly it is treated as a composite service containing one component service invocation.

Composite service requests create instances of workflows. Accordingly, if several composite requests arrive at the same time, they will be executed simultaneously.

Each client can invoke only one workflow and has an established SLA associated with it. Each workflow can have multiple clients. Clients issue requests at unknown times.

Each SLA contains a function reflecting the revenue generated by a successful composite service invocation. An invocation is considered successful if the response is obtained before some deadline (soft deadline d). In case processing of the request is finished after the soft deadline but before the hard deadline $\hat{d}$ the penalty p is applied to the service provider and the composite service invocation is considered unsuccessful. If at time $\hat{d}$ the composite request is still being processed, the workflow is terminated and is considered to be aborted. The abortion of a workflow entails a hard penalty $\hat{p}$ charged to the service provider. The abortion is not instantaneous as the workflow instance should wait for its component requests that currently being processed to complete.

The main objective of this research is to develop a scheduling discipline governing waiting queues in front of atomic services in order to maximize the profit of the composite services' provider. Scheduling policies also should meet a set of the following requirements:

1. **Sustainability**: Policies exhibiting the superior performance behaviour should also show sustainable and consistent results in boundary cases. In other words, in the worst

case their behaviour should be no worse than the behaviour of a corresponding naive policy or when no scheduling is used at all.

2. **Adaptivity**: Policies governing the system should be able to adapt automatically to changes in the environment. For example, after creating new workflows, the load on atomic services increases, which in turn leads to increased waiting times. The scheduler must detect this change and automatically adjust scheduling decisions to match the new conditions.

3. **Decentralization**: Often service workflows span services pertaining to multiple organizations each having a separate system administrator. Accordingly, the policies of the scheduler should require minimal amounts of centralized control, in the ideal case the control can be dispersed to a number of sites. Moreover, the scheduler should be able to perform scheduling without introducing any additional communication overhead.

4. **Deployment transparency**: Deployment of the scheduling solution should require minimal human intervention. Therefore, performing service code modification in order to implement a specific service is not an option. Consequently, the scheduling solution should be deployable to legacy services without modifying them.

5. **Platform independence**: The scheduling policy should not require any service providers to use any specific framework, OS or programming language for implementing and hosting services.

1.3 Overview of the Proposed Solution

The approach proposed in this thesis is based upon the notion of bi-level scheduling. The first layer takes as input an SLA and a workflows' structure and produces a scheduling context as output. The scheduling context contains parameters necessary for making scheduling decisions on components services' sites [Figure 1.7].

Three different global and four local scheduling policies are proposed. Each of them requires different amount of information and levels of control in order to be deployed. Moreover,

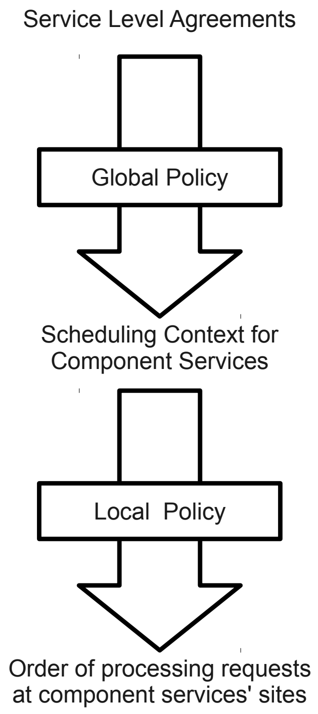
Service Level Agreements
Global Policy
Scheduling Context for Component Services
Local Policy
Order of processing requests at component services' sites

Figure 1.7: Bi-level scheduling

the proposed solution advocates the idea of decoupling global scheduling policies from local scheduling policies.

The proposed approach was evaluated by means of experiments employing simulations. The simulations were validated with experiments performed using a proxy implementing Static Priority scheduling. The experiments were conducted with two types of SLAs and two types of traffic.

1.4 Thesis Organization

The remainder of the thesis is organized in the following way:

- *Chapter 2* contains a literature review of the theoretical scheduling framework which is relevant to the addressed problem.

- *Chapter 3* contains a description of scheduling policies addressing the problem of meeting SLAs for composite web services. Namely, three different global scheduling policies and four local scheduling policies are described.

- *Chapter 4* comprises the evaluation of the proposed scheduling policies using a simulation. The simulator is calibrated to a real-world service tested with a non-preemptive priority scheduling policy. The remaining part of the chapter contains experiments conducted using the simulation.
- *Chapter 5* encompasses an outlook on the future research directions and a summary of the contribution.

Chapter 2

Background and Related Work

This chapter discusses related work done in the domains of scheduling theory, real-time systems and service-oriented systems which are relevant to the problem of meeting SLAs for composite services. The chapter contains four sections. The first section discusses scheduling in the context of service-oriented systems. The second section contains the overview of work focused on the problem of assuring SLAs for a single service. The third section focuses on the results of the work conducted in the context of composite services. The last section consists of the summary and the discussion of the related work.

2.1 Scheduling in SOA

In order to achieve full compliance with already stipulated SLAs, services should be managed in such a way that hosting resources are utilized efficiently and SLOs are not violated. In recent years methods of enforcing the QoS constraints for the service-oriented systems attracted a lot of attention. The work in this domain focused on dynamic discovery of services, dynamic service provisioning and, to a lesser extent on scheduling. Studies [4, 140, 3] show that dynamic discovery is an efficient method for ensuring the QoS of service compositions. The QoS requirements for workflows can be met by means of switching component services with the suitable QoS. However, sometimes component services cannot be easily switched at runtime as the switching process may require human intervention and can introduce significant latency, due to:

- Business constraints. The workflows are bound to certain businesses.
- Technical constraints. Alternative services can have different granularity, styles, WSDL

interfaces, etc.

- Limited alternative space. Certain services can offer a unique functionality, which has no alternatives.

Dynamic provisioning [95, 38, 7, 28] is a much more realistic approach which already has been used in practice. It is based upon dynamic allocation of resources to services according to changes in service demand. The services can be hosted on in-house available servers, or server capacities can be purchased from third-party providers, e.g. Amazon EC2[1]. Often hybrid schemes combining both approaches are employed as well. One of the most common ways of performing provisioning is based on past resource usage metrics, such as CPU, memory, IO, etc[2]. Such a method can be viewed as reactive. However, certain predictions regarding the coming workload can be made, therefore provisioning can be performed in a proactive manner [60]. Unfortunately, dynamic provisioning is not always possible and has a number of shortcomings:

- Limited scalability of legacy services: Modifying existing services in order to support horizontal scaling can require extensive service code re-engineering and often results in additional costs which may be quite significant and thus prohibitive. Some of the most common issues hindering transforming the legacy service code in order to support horizontal scaling are locks, static variables and service state management.

- Hosting limitations: A scenario where services cannot be scaled to accommodate the offered load, can occur when the company's rules require data to be physically present on the company premises due to privacy and security concerns. In such a case, the company would face the problem of implementing utility computing, for instance, as a private Cloud. Such an approach entails substantial additional expenses which may be too high for small and mid-scale businesses.

- Licensing restrictions: Licensing restrictions can be a limiting constraint. For example,

[1]Amazon Elastic Compute Cloud (Amazon EC2), `http://www.amazon.com/gp/browse.html?node=201590011`

[2]Amazon Elastic Load Balancing, `http://aws.amazon.com/elasticloadbalancing/`

certain software vendors charge per CPU fees and adding additional resources (CPUs) can be in violation with those license agreements.

- Provisioning latency: In case services can be scaled to accommodate the load, the issue of dealing with erratic unpredictable workload still persists, as adding additional resources (in case of an unexpected load jump) has certain latency associated with it (e.g. ten minutes with Amazon EC2). Consequently, the system would require that time in order to respond by allocating an additional amount of resources to services. Accordingly, during that time interval insufficient amounts of allocated resources can result in performance degradation. The performance degradation consequently would cause SLA violations.

- Limited SLA diversity: Dynamic provisioning by itself can be employed only when a service's clients' requests are subject to a single SLA, for example stipulating mean response time. However, if the clients have different SLAs and consequently QoS differentiation is required, dynamic provisioning by itself is not sufficient.

On the other hand, scheduling of incoming requests can be employed as a mean for ensuring differentiated Quality of Service and can serve as a complement to dynamic provisioning. For example, according to Ardagna et al. [8], QoS aware performance management of services spans (1) allocating resources for each service; (2) determining the volumes of traffic of requests for each service; and (3) scheduling policy at each server.

According to Zhou et al. [144] scheduling of requests in service-oriented systems might be conducted within three different sites: network, proxy and server [Figure 2.1]. The original classification is augmented by incorporating service specific components (e.g. virtual machines), due to the fact that a vast number of services are implemented to run on a virtual machine, e.g. JVM[3] or CLR[4].

The time required for delivering a message from a consumer to a service and back is defined as a network latency. Network latency can be managed by means of using network

[3]The Java Virtual Machine Specification, `http://java.sun.com/docs/books/jvms/`

[4]Common Language Runtime Overview, `http://msdn2.microsoft.com/en-us/library/ddk909ch.aspx`

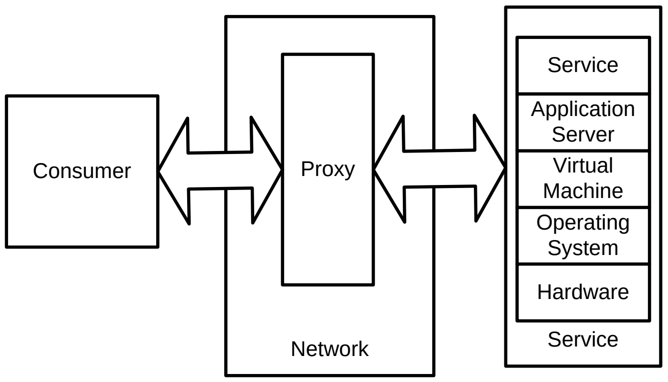

Figure 2.1: Typical service infrastructure.

packet scheduling which has already proven its efficiency on this domain. And as a result standards such as DIFFSERV [20] and RSVP [24]. have been developed and implemented by major network hardware manufacturers, e.g CISCO [5].

Scheduling on the proxy site implies sequencing the admission of jobs into a service. By means of a proxy it is fairly easy to intercept the incoming requests and schedule them due to the openness of the communication protocols (e.g. SOAP) advocated by the SOA principles. Generally speaking, scheduling on the proxy site enables decoupling a scheduling algorithm from a particular service implementation. Consequently, its deployment is not affected by the technical aspects of service, e.g. programming languages or underlying operating system.

Scheduling on the service site suggests performing scheduling in one of the following layers: operating system, virtual machine, application server and service's business logic. Generally speaking, scheduling of threads in a virtual machine is tightly coupled with the operating systems schedulers as Java threads (similar to managed .NET threads) are very likely to be executed within OS threads, i.e. one-to-one mapping is used. For example, in my previous work [47] scheduling of Java threads was performed by means of altering the priorities of the corresponding native threads.

At the same time, scheduling in the domain of operating systems has been a subject of intensive research [2]. However, only very recently, General Purpose Operating Systems, most commonly used by service community, were enhanced to support certain real-time features such as real-time threads [Section 2.2.3]. Scheduling of a virtual machine has already

[5]Cisco Systems, Inc., http://www.cisco.com

gained attention in the real-time research community. As a result, current virtual machine vendors offer solutions enabling basic real-time needs, e.g. real-time threads[6]. However, it is important to note that in order to take the full advantage of real-time properties of the virtual machine, the code needs to be engineered in a special way, as simply deploying existing services written in Java will not result into their augmentation with real-time properties, such as for example, service time determinism.

Scheduling within the application server/service's business logic suggests implementing services in a way that allows its scheduling. For instance, an application server can implement abortion or suspension of a currently processed request and returning system to a consistent state [100] or control locking process [41]. Even though, such an option appears to be appealing both from the logical and theoretical prospectives, deploying scheduling within the service's business logic itself would demand specifically designed services, which in its turn may result in high development costs.

The remaining part of the related work chapter contains three sections. In the first section the work addressing the problem of meeting SLAs for a single service are described. The second part contains the description of the studies done in the context of enforcing SLAs for composite services. The last section contains the discussion of the presented approaches. The presented approaches comprise relevant findings in the domain of scheduling theory, real-time systems, as well as studies of deploying scheduling in service-oriented systems.

2.2 Atomic Service Scheduling

This section contains an overview of studies conducted in the context of assuring response time timeliness of a single service. It covers the theoretical findings and the approaches deployed with service-oriented systems. The approaches are divided into two categories. The first category is focused on methods where service requests are scheduled before being admitted in service [Figure 2.2]. The second category of approaches are used in the scenarios where jobs accepted in service can be preempted (internal scheduling or scheduling of the threads handling the requests) [Figure 2.2].

[6]Sun Java Real-Time System, http://java.sun.com/javase/technologies/realtime/

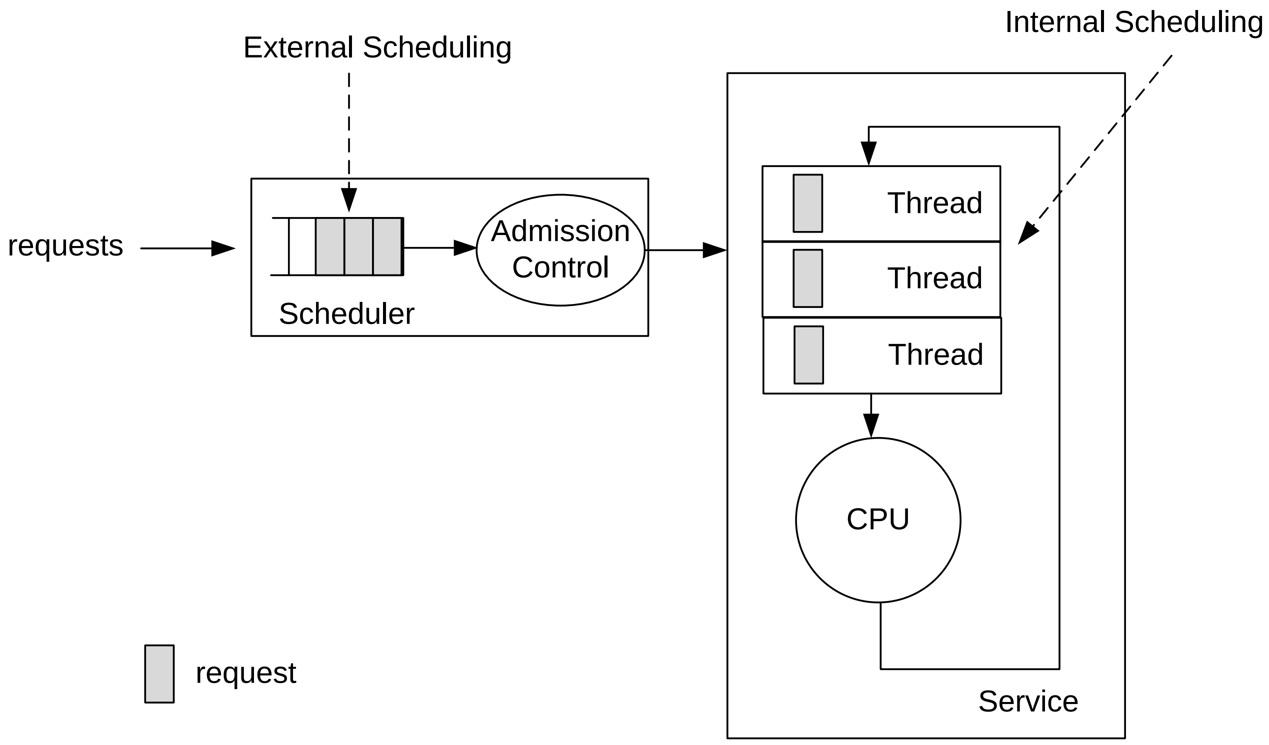

Figure 2.2: Non-preemptive (external), preemptive (internal) scheduling and admission control.

2.2.1 Admission Control

Currently, one of the most common architectures of application servers that hosts services proposes assigning each incoming request to a separate thread/process [1]. Each thread is being allocated a fraction of the CPU time and threads are being switched (preempted). The threads are sharing resources such as memory, IO, locks. Such a technique is used in order to improve utilization of the underlying hardware resources – while one thread is waiting for IO another can use the CPU. However, running too many threads at the same time is very likely to result in thrashing [39] due to resource or data contention [65]. The thrashing occurs due to excessively long waiting times when threads/processes are competing for the resources. At the same time, running too few threads/processes can result in insufficient resource utilization and affect the throughput of the system in a negative way [116]. Therefore, the service's efficiency of processing incoming requests strongly relies on the number of concurrent threads – the Multiprogramming Level (MPL). Accordingly, system administrators are facing the problem of finding the MPL leading to the best performance.

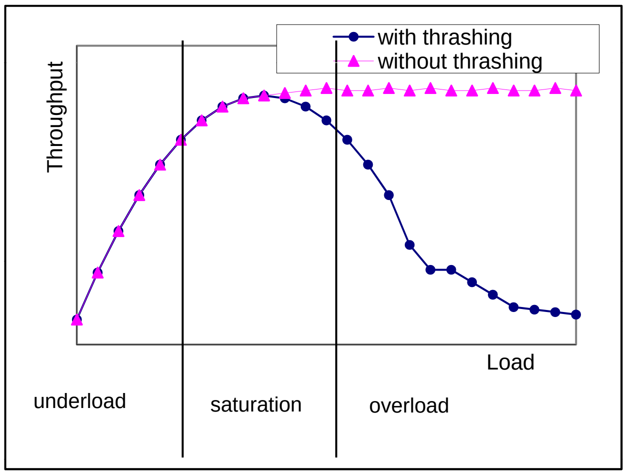

Figure 2.3: Throughput with respect to thrashing [45].

The problem of choosing the right value of the MPL was tackled by Heiss and Wagner [65]. The authors suggested two different approaches: incremental steps and parabola approximation. Other studies such as [111, 40] focused on implications of manipulating the MPL in order to achieve optimal response times. The authors compared two control theory approaches such as a Proportional-Integral (PI) [25] controller and a fuzzy logic controller. Diao et al. in [40] demonstrated the advantages of using a PI controller. Parekh et. al in [111] employed a fuzzy logic controller and compared it to a PI controller [25]. The fuzzy logic controller exhibited higher response time than a fine application tuned PI controller, yet it required much less human intervention.

The key roles of the MPL are protecting the system from an overload and maximizing its throughput. However, it can also be used as a means for controlling the length of the waiting queue. Schroeder et al. [116] demonstrated that there is a strong dependence of the queue size on the MPL; the higher the MPL the shorter is the waiting queue, and consequently the shorter the waiting time. Besides that, the more service requests are admitted in the service the higher is the service time [Figure 2.4]. The authors argue that in the case of higher values of the MPL, reordering the requests in the queue has a lesser impact on the response time, as non-preemptive scheduling affects only the waiting time of the requests. Essentially, if the MPL is too high, then an external scheduler will have little to no impact, as there will be no requests in the waiting queue. Accordingly, the authors advocate the idea that using a lower than optimal MPL does not lead to the highest throughput. However, it enables better

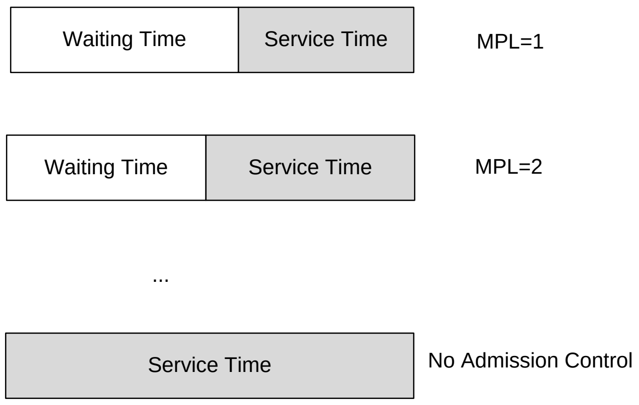

Figure 2.4: Waiting and service times dependence from the MPL.

control over the flow of incoming requests which subsequently gives better control over the response time.

Another drawback of accepting an unlimited number of jobs and handling them according to the Processor Sharing policy, is apparent when services are combined into chains and are exposed to request arrivals in bursts [42, 46]. The burst arriving at the first service tends to condense and propagate over the chain, resulting in a dramatic performance degradation (workflow response time increase). The mentioned situation can be mitigated or completely eliminated by means of exercising admission control, restricting the number of concurrently processed jobs.

Elnikety et al. demonstrated the efficiency of admission control deployed on e-commerce web sites [50]. The authors suggested scheduling a waiting queue according to the Shortest Job First scheduling policy. As a result the thrashing effect was eliminated and the response time was optimized. Note that all work cited in Section 2.2.4 have employed some form of admission control.

2.2.2 Theoretical Background

This subsection contains a description of the most relevant findings in the area of scheduling theory. There are three different types of scheduling: deterministic, stochastic and online. In deterministic setups the complete and precise information on job arrivals and job sizes are known ahead of time, while in the stochastic case usually only distributions and their parameters of the aforementioned values are available. Online scheduling addresses the scenarios where job arrivals and/or their sizes are not known ahead of time. The next subsection provides the theoretical perspective on the problem addressed in this thesis and therefore is necessary for understanding the complexity of the problem in terms of the computational costs.

Deterministic Scheduling

Problems in offline deterministic scheduling are closely related to combinatorial problems. Therefore, the research community has mostly focused on the problems having polynomial solutions or proving that problems are NP-hard [86, 26, 56, 22].

In order to simplify and formalize the future description of the reviewed scheduling techniques, the following notation will be used: r - arrival time or release time, p - processing time or job size, C - completion time, d - deadline. In order to reuse the legacy terminology used in offline deterministic scheduling, services will be referred to as machines, service requests as jobs, and component services invocations used for handling composite service requests as operations.

In order to systematize the space of scheduling problems, the three-field classification ?/?/?, introduced by Graham in [64], is employed. The first parameter describes the number of serving machines or service replicas[7]. The second parameter refers to the job characteristics [Table 2.1], and the last parameter denotes the optimality criterion.

Table 2.1: Scheduling problems abbreviations.

Parameter	Meaning
$pmtn$	Jobs can be preempted
$prec$	There are precedence relations between jobs (operations in a job)
r_j	Jobs have arrival times other than zero ($r_j \neq 0$)
$p_j = p$	All jobs have equal sizes p

Each scheduling problem is subject to minimizing an objective function $f()$ associated with it. The objective function describes the value of completing jobs at specific times,

[7]It can also describe the configuration of the machines as well as the relations between jobs.

e.g. number of jobs overdue. The schedule A is *optimal* if $f(A)$ can not be further reduced [26, 35, 56]. Commonly used functions are: lateness $L_j = C_j - d_j$, earliness $E_j = max(0, d_j - C_j)$, tardiness $T_j = max(0, C_j - d_j)$, flow time $F_j = C_j - r_j$, unit penalty $U_j = \begin{cases} 0 & C_j < d_j \\ 1 & C_j \geq d_j \end{cases}$.

Problems containing minimization of a total flow time represent problems of the overall QoS optimization (service response time improvement). These problems are important in scenarios where SLAs have the same QoS constraints and optimizing QoS provides the possibility to accept more clients into the system [104, 103, 43]. Tardiness, lateness and especially unit penalty refer to other SLA constraints imposing limitations on the deviation of the response time from a service, or limiting the number of deadline violations [44, 49].

Another important characteristic is the number of classes of contracting clients; it is either a single class or multiple classes. For single class clients the objective function will most likely be represented as a sum or maximum. In case the multiple clients/contracts are presented, the weighted sum should be used (w_jT_j), as the "value" of the clients should be incorporated into the objective function. The presence of the economic model, explicitly "defines" the importance of each consumer, since it determines the revenues brought by successful invocations and penalties incurred for SLAs infringements.

$1|prec; p_j = p; r_j|L_{max}$ was solved by Simmons [123] in 1978. A couple of years later, Simmons [122] proposed a solution for $1|prec; p_j = p; r_j|\sum C_j$. However, Baptiste found a polynomial solution to the more general case $1|prec; p_j = p; r_j|\sum w_jC_j$ [15]. One of the key problems directly corresponding to assuring SLAs, is the weighted unit penalty problem. $1|p_j = p; r_j|\sum w_jU_j$ has a polynomial solution according to Baptiste [14]. At the same time, $1|p_j = 1; r_j|\sum w_jT_j$ appears to be an assignment problem [26]. In [81], Lawler suggested using an iterative algorithm for solving $1|prec; r_j|C_{max}$. The algorithm constructs the schedule from the back to the front and requires $O(n^2)$ steps. It is important to note that, in the original paper, Lawler's algorithm minimizes the maximum of non-increasing function, e.g. unit penalty or tardiness.

Problems $1||\sum w_jTj$ and $1||\sum w_jU_j$ are of interest as they refer to the scenario where SLAs are based on deadline violations and weight is corresponding to the business values of requests. Unfortunately, many scheduling problems appear to be NP-hard: Lenstra et al. [85] proved that $1|r_j|\sum C_j$ is NP-hard and therefore all of its superclass problems ($1|r_j|\sum w_jC_j$,$1|r_j|\sum T_j$,$1|r_j|\sum w_jT_j$,) are also NP-hard. Two of the most valuable results were obtained by Lawler and Moore [84], as well as by Karp [74]. They proved that $1||\sum w_jU_j$ is NP-hard. Lawler [82], and Lenstra et. al. [85] demonstrated that $1||\sum w_jTj$ belongs to the NP-hard problems.

2.2.3 Internal (Preemptive) Scheduling

As it was mentioned before a de facto model suggests handling incoming requests in separate threads. The process of selecting the next thread/process to be executed on a CPU is referred to as thread/process scheduling.

Lawler [83] showed that $1|r_j;pmtn|\sum U_j$ has a polynomial solution, while $1|r_j;pmtn|\sum w_jU_j$ is NP-hard. $1|pmtn;p_j=p;r_j|\sum w_jU_j$ also has a polynomial solution [17]. $1|prec;pmtn;p_j=p;r_j|\sum C_j$ problem according to Baptiste et. al [16] does not require more than polynomial time to be solved. The problem $1|r_j,pmtn|\sum U_j$ has a polynomial solution [83], while the problem of $1|r_j,pmtn|\sum w_jU_j$ does not have a polynomial solution [83].

The most closely related area in this case would be real-time scheduling. The most typical problem in this domain is scheduling of jobs of a cyclic nature. If a task consists of jobs arriving with the exact time period, the system is called periodic. The system is called sporadic if only the minimum time between consequent jobs in a task is known. Generally speaking, in the real-time scheduling domain there are two main types of scheduling algorithms; that is dynamic priority and static (fixed) priority. Dynamic scheduling refers to the algorithms which alter the priority of a task's job at runtime [86].

Real-time systems can be divided into two classes: soft-real time and hard real-time. In soft real-time systems it is allowed to miss a certain ratio of the deadlines, while in the hard real-time systems the system is considered failed if it misses even one deadline. Hence, in the context of hard real-time systems one of the key questions is determining the schedulability of a system – the ability of a system to meet all the deadlines.

Static Scheduling

Static scheduling refers to the algorithms which assign priorities to the jobs in an offline manner and the priorities remain unchanged at runtime. One of the key static scheduling approaches is the Rate Monotonic Scheduling (RMS) [90] which addresses the problem of meeting the deadlines for periodic jobs. RMS suggests sequencing jobs according to their periods, thus the jobs arriving at higher rates will have a priority over jobs arriving at lower rates. Liu and Layland [90] investigated the least upper utilization bound under which RMS can meet all the deadlines in case deadlines are equal to the periods. The authors showed that the system will be schedulable if the worst case utilization $\sum_{j=1}^{n} \frac{p_j}{T_j}$ of preemptive machine does not exceed $n(2^{1/n}-1)$, where n is the number of periodic tasks and T_j denotes the period of request arrivals within the task j. Therefore, for one task the utilization can reach 100%, for two 83%, and 69% if the number of tasks approaches infinity. This is only a sufficient condition, which is somewhat pessimistic. Nonetheless, Liu and Layland consider the worst case scenario, when all tasks have the same release time, and in other cases the boundaries can be increased [87].

Other static priority policies include Highest Value First [29] and Highest Density First [29]. Highest Value First (Static Priority) refers to the policy suggesting handling the incoming requests in the order of their (business) value [29]. Highest Density First (HDF) prioritizes the requests according to their density, which reflects the amount of value obtained for processing a time unit of a request. The last is expressed as follows: w_j/p_j [29].

Dynamic Scheduling

One of the best known dynamic scheduling algorithms is Earliest Deadline First (EDF). If a feasible scheduling sequence exists, EDF is guaranteed to find the optimal solution [90]. However, the problem of answering the question regarding the existence of a feasible solution for environments with periodic tasks does not appear to have a polynomial solution [86].

The method of calculating utilization in order to determine schedulability of systems governed by RMS is not sufficient to be used with EDF. Therefore, a more formal definition for sporadic loads was introduced by Baruah et al. in [18]. "A sporadic real-time environment is said to have a loading factor b if and only if it is guaranteed that there will be no interval of time $[t_x; t_y)$ such that the sums of the execution-times of all task-requests making requests and having deadlines within this interval is greater than $b(t_y - t_x)$". Baruah et al. state that the system governed by EDF is schedulable only if its b-factor is no larger than 1. Such an approach was successfully used with industrial service-oriented applications where EDF was used for ensuring the CPU resource partitioning between service clients [37, 106].

One of the major shortcomings of EDF becomes evident in the setting where the job arrivals are not deterministic, i.e. schedulability analysis cannot be performed. In such cases overload situations can emerge and unfortunately EDF cannot provide any guarantees on the tasks which will fail as a result of the "domino effect" [93]. The "domino effect" usually emerges in transient overload situations (b-factor greater than 1 [18]), e.g. occurring due to burst arrivals. In this case certain jobs cannot meet the deadlines but are still being processed. As a result, the effective utilization of the machine is even more decreased, and more subsequent jobs are becoming overdue. In the worst case, the efficiency of EDF can approach zero as all the jobs will be completed after the deadlines [93]. In order to avoid the "domino effect", "guaranteed" and "robust" admission control mechanisms were developed [29, 30, 31]. Both methods attempt to make a prediction regarding schedulability of newly arrived jobs. The "guarantee" mechanisms suggest inserting a newly arrived request in a waiting queue and checking if the jobs in the waiting queue can be completed by their deadlines; if not they are rejected. The "robust" modifications propose adding a new job to the queue, and in case there is a job which can not be processed before its deadline, the least valued jobs will be exempted from the queue. Every time job processing is completed, the most valued task from the set of the exempted jobs (with a positive laxity) is selected and an attempt to reaccept the job is made. Laxity y is determined as follows: $y_j = d_j - (t + p_j)$, where t is the current time and the second item refers to the expected completion time if the job processing would have been started at time t. Robust EDF (REDF) [29, 30, 31] was successfully deployed in [124].

Minimum Laxity First (MLF) suggests reordering jobs in the non-increasing order according to their laxity (or slack time). Essentially, laxity reflects the amount of time the job can be delayed and the deadline can still be met. Obviously, jobs having laxity lesser or equal to zero have no chances of being processed in time. Uthaisombut [131] has extended the idea of MLF by introducing Compound Least Laxity First scheduling policy. The algorithm suggests sequencing requests according to the compound laxity - the maximum time for delaying the job so it will not miss its deadline, while considering the presence of the other jobs.

The group of the algorithms mentioned above (except REDF) comprises the methods of meeting the deadlines in case the users are treated equally. However, in the context of SLA, the users have a business value associated with each client. Thus, each request besides the d_j, will have a value of w_j – which for instance – can represent the service fee charged for each successful service invocation [29].

Maximum-Urgency-First (MUF) belongs to the group of mixed algorithms as it combines static and dynamic scheduling. Each priority (urgency) consists of the following attributes: *criticality, dynamic priority and user priority.* The method orders the incoming jobs in accordance their attributes: criticality, dynamic priority and user priority. *Criticality* is a static priority assigned according to the arrival rate of jobs of each task, the tasks with the arrival rates inducing 100% utilization in the worst case are marked as the jobs with the high criticality, while the rest will have low criticality value. The *dynamic priority* is inversely proportional to the laxity, and *user priority* indicates the importance (business value) of each job. The scheduling policies mentioned above have been successfully deployed with many real-time CORBA services [59, 36].

Comparative studies between EDF (REDF), HDF and HVF have showed that REDF and HDF admission control (Robust HDF) performs the best in most of the overload situations, while in underload cases (if there is a feasible schedule) EDF exhibits better results [29].

Utility Accrual Scheduling

All previously discussed approaches are targeting optimizing the number of late jobs. However, there can also be situations when a service is exposed with different SLA types at the same time. For example, one class of clients can contract a service with an SLA according to which a penalty depends on the number of late jobs, while another class of clients has an agreement with the service provider, where the penalty depends on the tardiness of requests. Consequently, the service provider is facing a problem of meeting two criteria at the

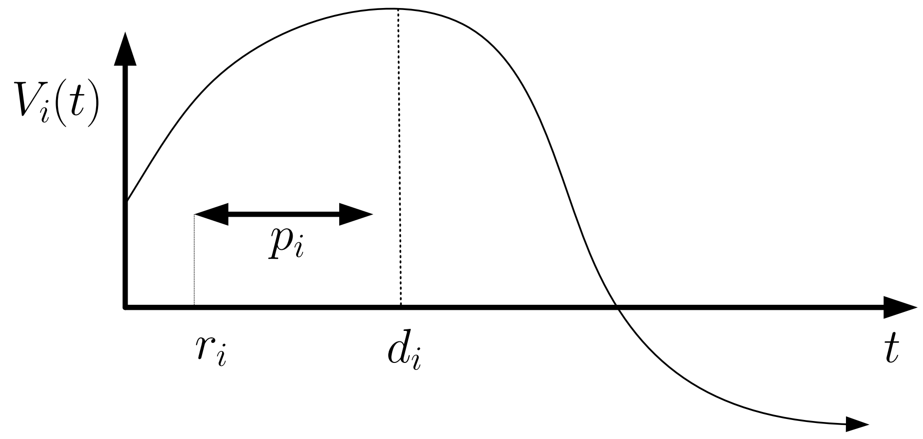

Figure 2.5: Total Utility Function (TUF).

same time, weighted sum of unit penalties $\sum w_j U_j$ for the first class and weighted sum of tardiness $\sum w_j T_j$ for the second class. Jensen [70] suggested tackling this problem by means of exercising a value function. The latter indicates the benefits of completing a job within a specific time. An example of such a function can be seen in [Figure 2.5[8]], where V is the "value" rewarded for completing the job at time t. Another widely adopted term for the value function is time/utility function (TUF). By means of TUF it becomes possible to specify soft deadlines along with the hard deadlines for the same machine. Despite the fact that TUF was originally designed for scheduling processes in real-time operating systems [70], it was also used with real-time CORBA middleware [88].

At present, there is a set of various methods developed and used for scheduling jobs having a TUF function on a single machine. The simplest scheduling policy is Earliest Deadline First [70].

Another utility accrual scheduling approach is D^{over} [79]. D^{over} schedules jobs according to their Latest Start Times, defined as the deadline minus the remaining job size. The algorithm is enhanced with a special heuristic mechanism which allows detecting jobs that are going to be late and rejecting them based on their values, which makes this approach superior to MLF.

Locke's Best Effort Scheduling Algorithm (LBESA) [93] is one of the earliest works at-

[8]TUF may increase in case a request should be completed as close as possible to its deadline, yet completing it earlier is not desirable. Such a scenario might occur, for example when the service represents some equipment, e.g. a radar system, and the client needs to obtain the samples at the exact times.

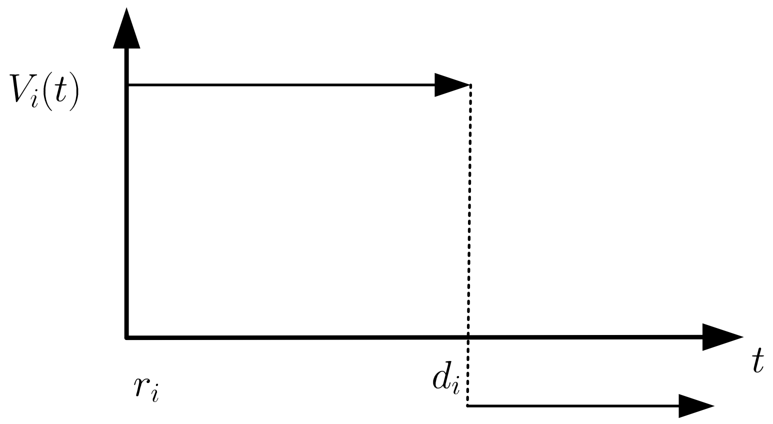

Figure 2.6: Step TUF.

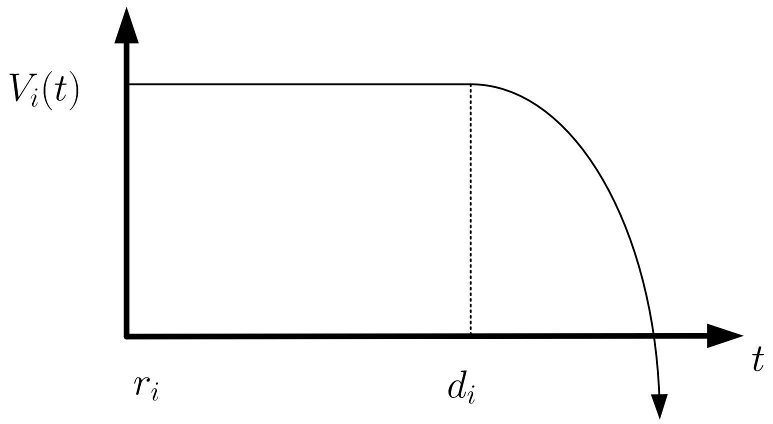

Figure 2.7: Left concave TUF.

tempting to maximize an accrual utility function of an arbitrary shape. The algorithm suggests using EDF in case of underload situations, and rejecting certain jobs when an overload occurs. The rejection decision is based entirely on the Potential Utility Density (PUD) function value, which refers to the amount of possible utility over time the system can be rewarded. The jobs with the smallest PUD values are rejected first. In case all the remaining jobs can meet their deadlines, they are ordered in the non-increasing order according to their PUD. The algorithm does not consider any operation precedence constraints, and thus can be used only as a governing policy for a single machine handling independent jobs.

Clark [34] has described a Dependent Activity Scheduling Algorithm (DASA). In contrast to LBESA, Clark considers the operations' dependencies, but the systems are still restricted to a single machine. It schedules the operations according to the EDF if the system is underloaded and rejects tasks on the basis of their PUDs. However, the rejected operations are sorted in a decreasing order of PUDs and then one by one they are inserted in the tentative schedule. If the last appears unfeasible, the operation is rejected and the next one is attempted to be scheduled. The algorithm is limited to the step TUFs [Figure 2.6].

Another alternative is the Generic Utility Scheduling Algorithm (GUS) [89], which can be exercised as a method of scheduling dependent threads with locks in order to maximize the accrual utility. GUS schedules the threads according to PUD. At runtime, GUS selects the threads(operations) according to the "greedy" strategy which suggests choosing the thread with the highest PUD as it would yield the maximum accrual utility value.

Job Preemption in Concurrent Systems With Locks

The typical architecture of a service implies handling each request in a separate thread of execution[9]. The default setting of application servers hosting services suggests treating each handling thread in a fair manner, by assigning them equal slices of the CPU. In order to give priority to certain jobs, the priority of the corresponding handling threads has to be increased.

If threads are not using any synchronization, such as semaphores, mutexes, monitors

[9].NET Remoting Overview - `http://msdn2.microsoft.com/en-us/library/kwdt6w2k.aspx`, Apache Axis - `http://ws.apache.org/axis/`

and neither IO, the preemption can be implementated relatively easy. However, in case the threads are using synchronization, a priority inversion problem can occur [119]. For instance, job i has to be executed at the highest priority, but it also requires resource x (e.g. lock) which is currently occupied by job j, running at a lower priority. Consequently, i needs to wait for j to release the lock and as a result, the execution time for i significantly increases.

One of the key methods, addressing the priority inversion problem, proposes employing the Priority Inheritance Protocol [119]. The Priority Inheritance Protocol consists of raising the priority of the thread currently possessing the resource to the priority of the thread waiting to acquire the resource. In such a way, this method ensures that the waiting time for the thread with the highest priority is minimized. Essentially, the Priority Inheritance Protocol introduces a push through blocking on top of the existing direct blocking. Although basic Priority Inheritance has a rather straightforward implementation, its deployment can be hindered due to a chance of deadlock or chained blocking [119]. Nonetheless, it was successfully deployed in [100] where McWherter that Priority Inheritance protocol showed its supremacy over raising priorities of handling threads which employ direct blocking. At the same time, Schroeder et al. [116] showed that using the performance of an admission control combined with non-preemptive scheduling can approach the performance exhibited by preemptive prioritization combined with the Priority Inheritance Protocol in the cases when a Web Service implementation employs a large number of locks.

Another alternative is the Priority Ceiling Protocol [119], which reduces the system's vulnerability to deadlocks and offers a remedy for the chained blocking. The algorithm extends the Priority Inheritance Protocol by introducing the notion of priority ceiling. The priority ceiling refers to the requirement on the priority of a job, allowing entering the corresponding critical section (acquiring the resource) only in case a thread's priority exceeds the ceiling value. Unfortunately, deployment of the Priority Ceiling Protocol requires a priori knowledge on the probable required locks for each possible job [119]. Moreover, it introduces an additional overhead in comparison to the Priority Inheritance Protocol [119].

A modification of the Priority Ceiling Protocol, namely the Distributed Priority Ceiling Protocol is implemented in real-time CORBA [36]. Ding investigated the potential of applying the Priority Ceiling Protocol to J2EE application servers in [41].

In contrast to the Priority Ceiling Protocol, the Stack Resource Policy, introduced by Baker in [13, 12], reduces the number of context switches, and is designed to be directly deployed with dynamic (EDF) and static scheduling.

Preemptive real-time scheduling can be conducted only on the service side and demands real-time OS[10] or real-time virtual machines[11]. Not only do real world applications comprise CPU bound operations, but they also comprise various interaction with hardware, where the preemption is difficult or too expensive to implement [77]. Consequently, performing external (non-preemptive) scheduling is often the only viable option.

Unfortunately, in practice only simple scheduling techniques, such as priority based scheduling, are supported by General Purpose Operating Systems such as Linux. And for example, EDF support was added to the Linux operating system kernel [53] only in 2009.

2.2.4 External (Non-preemptive) Scheduling

External scheduling refers to reordering requests before they are admitted in service.

One of the key theoretical results is Moores-Hodgson's algorithm which solves $1||\sum U_i$ in polynomial time, specifically in $O(n \log(n))$ [102]. The core idea of this approach is the ordering of the jobs in the Earliest Deadline First order and then elaborating the schedule by first finding the first job σ which is late and removing the largest job from the set of preceding σ jobs.

Lawler and Moore [84] showed that $1||\sum w_i U_i$ can be solved by means of using dynamic programming and thus, the proposed solution is pseudo-polynomial with the complexity of $O(n \sum p_j)$, or NP-hard in a weak sense. Note that the same solution also can be applied to parallel machines.

For scenarios not permitting preemption, the problem with available release dates $1|r_j|\sum U_j$ appears to be NP-hard in a strong sense, as it is reducible to $1|r_j|L_{max}$, which was shown to be NP-hard in a strong sense by Lenstra in [85]. Subsequently $1|r_j|\sum U_j$ is also NP-hard in a strong sense.

[10]Sun Solaris http://www.sun.com/software/solaris/index.jsp, QNX http://www.qnx.com/
[11]Real-time JVM, http://java.sun.com/javase/technologies/realtime/

Non-preemptive Accrual Scheduling

Non-preemptive accrual scheduling refers to the class of non-preemptive problems aiming at maximizing the accrual utility.

Yu et al. [139] addressed the problem of maximizing the revenue, where job arrival times are unknown; however the distribution of the job sizes, as well as their maximum and minimum values are available. The proposed algorithm called Revenue and Penalty Aware (PP-aware) performs scheduling of a single processor and has been shown to be superior to other accrual schedulers such as EDF and "greedy" PUD based scheduling (employing the principle used by GUS yet in a setting where preemption is not allowed).

Liu et al. in [91] extended the classical model, by introducing a penalty for a job rejection. Note that a penalty for a job abortion also depends on the time of rejection. The non-preemptive scheduling algorithm proposed by the authors has been shown to be superior to EDF, GUS (non-preemptive version) and PP-aware.

Service Resource Sharing by Means of Generalised Processor Sharing Scheduling

While the previously described algorithms tackle the problem of enforcing the deadlines for particular request processing, another branch of work focuses on sharing service capacities in a manner allowing enforcing other types of SLA, such as mean response time. This is being achieved by means of employing the Generalized Processor Sharing policy (GPS). GPS enforces capacity sharing by allocating some resource share ϕ_k for each client class k. Clients' resource shares determine the service throughput seen by each client, which consequently reflects on their response times. Essentially, GPS is based upon the idea of dividing a processor into virtual pieces. The response time guarantees can be given only with certain probability which can be found analytically [110].

GPS [Figure 2.8] refers to an abstract policy where requests have infinitesimal size to certain class (client) k and each class is allocated some resource share ϕ_k [110]. Such a model is called fluid and does not exist in reality. Therefore other methods are used for approximating it, such as Weighted Fair Queueing (WFQ) [19], Packet GPS (PGPS) [110, 143], Start-time Fair Queueing (SFQ) [62, 63], Controlled Start-time Fair Queueing (C-SFQ(D)) [73], Worst-

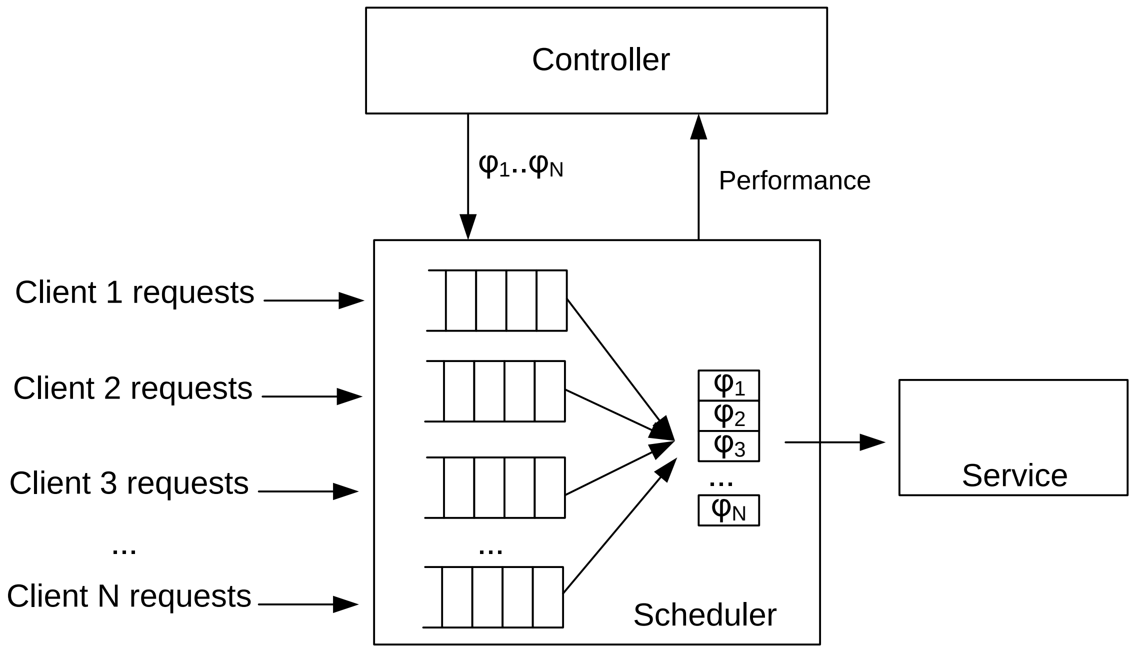

Figure 2.8: Non-preemptive Generalized Processor Sharing architecture.

Case Weighted Fair Queueing (WF^{2}Q) [19] and many others.

One of the most interesting properties of the GPS scheduler is the dependence of the waiting time on the resource share. This property is also known as work conservation law [78]. Work conservation law states that if each class of clients is allocated some resource share ϕ_i, ($\sum_{i=1}^{N} \phi_i = 1$), the average delay time W_i for each class i follows the rule: $\sum_{i=0}^{N} \phi_i W_i = const.$ Therefore, the average response time can be expressed as $F_i = p_i + W_i$.

Finding the resource shares can be performed in an open-loop or a closed-loop way. In the first case, the main objective is finding the values of ϕ. The initial resource shares are normally found when the share of the system seen by each client is approximated as a single processing server and then queueing theory analysis is applied [142, 92].

A closed-loop method implies that the resource shares are continuously adjusted, e.g by using a controller. In such a case admission rates for each class of clients is the control and response time is the measured output [Figure 2.8] [94, 73].

One of the key issues when deploying GPS in practice is achieving the proportional throughput with accordance to the corresponding resource shares ϕ. The "fairness" of those scheduling policies is determined by the proximity of the ratio of the achieved throughput seen by client classes over the total throughput to the admission rates. As a result a number of policies addressed the issue of achieving "fairness", e.g. Weighted Fair Queueing (WFQ) [19], Packet GPS (PGPS) [110, 143], Start-time Fair Queueing (SFQ) [62, 63], Controlled Start-time Fair Queueing (C-SFQ(D)) [73], Worst-Case Weighted Fair Queueing (WF^{2}Q) [19] and many others.

Liu et al. [92] employed Weighted Fair Queueing for scheduling a set of machines hosting a set of services (server farm). Each service is offered within different classes. The SLA stipulates the maximum percentage of responses with response times greater than a predefined value. The authors addressed the problem of routing the requests (assigning the flows) to the services hosted on particular machines, where the arrival rates, service configurations, machine capacities and SLAs are given. Liu et al. showed that the problem can be reduced to a network flow model with a set of concave objective functions. The offered solution drastically outperforms the naive proportional allocation approach.

An application of Weighted Round Robin on the service side has been investigated by Menascé et al. [101]. The authors addressed the scenario where jobs had a business value determined by parameters of service requests.

Lottery scheduling [136] also targets overcoming the issue of starvation. Essentially, each request residing in a waiting queue is being assigned a certain number of "lottery" tickets. Each time the service becomes available for accepting another request, the "lottery" takes place. After that the winner is forwarded to the service and processed. This method was used as a means for assuring differentiated throughput under fluctuating client loads and implemented for Web Service middleware prototypes [120, 121]. Experiments showed the viability of this approach and its superiority over plain static priority assignment. It is important to note that, due to its non-deterministic nature, lottery scheduling results in geometrically distributed waiting time. In recent works the authors elaborated the method by introducing stride scheduling which enable lowering the variance of waiting time by several orders of magnitude [136] as well as the throughput error.

Nonetheless, when using GPS derivatives it is fairly easy to find the probability of the response time as being below a given value. Most of these analytical tools were derived for scenarios where it is assumed that job arrivals are Poisson and of the certain rate λ [66]. However, there are approximations which can relax these requirements [138].

Another significant shortcoming of using GPS policies can be seen in situations where traffic is not stationary but changes over time. In such cases, reactive changes to the load fluctuations entail additional latency in detecting load changes and adjusting the admission rates [94]. Therefore, defenders of this approach actively advocate the idea of dynamic resource provisioning [66]. The latter can be done is a reactive manner responding to the load changes or in a proactive manner by foreseeing the future trends in loads changes. Forecasting tools range from simply using the last observed value of the arrival rate to more complex ones which also factor in seasonal and trend components of the traffic [60, 133].

2.3 Composite Service Scheduling

Composite services refer to services that employ other services in order to create new functionality. For example, an on-line store service can employ shipping service for generating and printing shipping labels and credit card service for handling money transfers [129]. Like atomic services, composite service can be subject to SLAs. This subsection describes a list of theoretical and practical approaches addressing the problem of meeting SLAs for service workflows.

2.3.1 Theoretical Background

In the offline deterministic domain workflow scheduling corresponds to the job shop scheduling problem. The problem is annotated with letter J and refers to a setting where each job consists of operations having precedence constraints. Usually, the operation relations are expressed in a form of Directed Acyclic Graph (DAG). Each operation can be processed only on a dedicated machine. A special case of the job shop problem is a flow shop problem F, where the precedence relations are forming a chain. Unfortunately, in this domain very few problems have polynomial time solutions. The two machine flow shop problem $F2||C_{max}$ was solved by Jackson and has an $O(n \log n)$ solution [69], which is essentially a generalization of Johnson's rule [71]. While other problems such as $F2|p_{ij} = 1; chains|\sum w_iC_i$ $F3|p_{ij} = 1; chains|\sum w_iC_i$ [126] $F2|p_{ij} = 1; chains|\sum U_i$, $F3|p_{ij} = 1; chains|\sum U_i$ $F2|p_{ij} = 1; chains|\sum T_i$ and $F3|p_{ij} = 1; chains|\sum T_i$ [27], are already NP-hard.

2.3.2 Heuristic Approaches

Due to the inherent complexity of job shop (workflow) problems, they are often addressed by heuristics. Generally speaking, heuristics are a way of expressing a rule of thumb more formally [58]. For heuristics, it is a common practice to aggregate certain characteristics of jobs and express them in the form of a priority. Apart from their simplicity, heuristics are also extremely attractive from the implementation perspective as they can have extremely low computational overhead.

Fortunately, job shop heuristic scheduling has gained a lot of attention within the research community from the manufacture/production scheduling domain. However, there are some peculiarities that make workflow scheduling different from job shop scheduling in manufacturing, where most focus is given to the problems of job shop scheduling with a common due date $d_j = d$ [61] or minimizing the makespan C_{max} [68]. In the context of service-oriented applications, the more interesting question is minimizing $\sum T_i$ (sum of tardiness), $\sum w_i T_i$ (sum of weighted tardiness). Besides that, in manufacturing it is a common assumption that the precedence relations are forming a chain structure [107, 113] or in terms of workflows, they are composed only according to the sequence pattern [132]. Consequently, the precedence relations are static. However, in certain service workflows, component services can be invoked in parallel (exclusively, looped) and thus, the precedence relations can be dynamic [32, 33]. One of the common ways of tackling this issue is an adoption of reduction methods, which would enable interpreting the dynamic precedence relations as static [32, 33].

A survey done by Baker [11] shows that there is no single heuristic demonstrating the superior results in all cases, as they exhibit various behaviour depending on evaluation contexts. Therefore, this work lists only the approaches which have shown the best results at least under certain experimental conditions. First of all, some additional notation should be introduced. Denote allowance as $a_j = d_j - t$, slack time as $s_j = a_j - p_j$, where t is the current time and p_j refers to the size of the job itself.

The most promising policies include (for a comprehensive list of job shop heuristics analysis papers check [11]):

- Minimal Slack Time (MST): The operations of a job are ordered in the nondecreasing order of s_j.

- Allowance Per Operation (A/OPN): The operations are ordered according to a_j/k_j, there k_j refers to the number of remaining unprocessed operations.
- Slack Per Operation (S/OPN): The operations are ordered according to s_j/k_j.
- Smallest Critical Ratio (SCR): The sequence of processing is determined by a_j/p_j.
- Proportional S/OPN: Allowance per operation, the operations are ordered in accordance with s_j/p_j.
- Shortest Job First (SJF): The policy is a non-preemptive scheduling policy which suggests ordering the requests according to the job sizes: the smaller jobs are put in front of the bigger jobs. SJF is proven to be optimal for the $1|p_j|\sum F_j$ problem [35]. Nonetheless, surprisingly SJF showed strong results without directly addressing the timeliness of jobs [21].

It is important to note that other heuristics are also possible. For instance, in work [107] Panwalkar and Iskander described over one hundred policies.

2.3.3 Bi-level Scheduling

One of the key dimensions characterizing heuristics is the scope of scheduling. Scheduling can be conducted on a component request (operation) level or on a workflow (job) level. Operation scope implies decomposing the workflow QoS requirements (deadlines) into the deadlines for each operation and then sequencing operations, e.g. according to EDF.

For workflows aggregating component services in a sequence structure, the deadline can be decomposed into component requests deadlines in the following ways:

- Using the global deadline $d_{ij} = d_j$,[72, 11, 44]
- The time between arrival time and deadline can be divided between operations
  - Equally (Equal Flexibility (EQF)) [72, 11]
  - Proportionally to operation sizes $p_{i,j}$ (Proportional Flexibility (PQF))[72, 11]

In the domain of rewarding algorithms, this technique is denoted as TUF decomposition. At present, a number of various methods has been proposed [88] and analyzed:

- Ultimate TUF(UT): The global TUF is utilized as a local TUF for each component.
- Scaling based on EQF (SCEQF) [72]: The global deadline is decomposed into local deadlines according to the mean service times. The TUF height is downscaled in the same manner.
- Scaling based on TUF shape (SCALL) [88] : This algorithm operates in a similar way to the last one. However, there is a set of differences. First of all, this method assumes that the value function is unimodal (has only one peak). Secondly, if the job can be completed before the time corresponding to the maximum value of the TUF, the function is truncated in such a way that the new deadline is at the optimum point.
- Decomposing into linear-constant TUF (OPTCON)[88] : The method proposes changing the shape of sub-TUFs in contrast to SCEQF and SCALL which suggest slicing the deadline. OPTCON performs the TUF decomposition according to the following rules.
  - The deadline for all operations is equal to the deadlines of the job.
  - The utility linearly increases from the arrival time to the estimated completion time. The TUF maintains a constant value equal to the maximum one at time of the estimated completion till the deadline.
  - The TUF for the operations is downscaled by a factor which corresponds to the ratio of the PUD for the operations (local threads) over peak utility density which represents the ratio of the optimum value over the expected execution time.

The experiments conducted by Li [88] showed that OPTCON and TUFS exhibit the superior behavior with the PUD based algorithms (GUS, DASA, LBESA), at the same time SCEQF and SCALL demonstrate the best results when used with the deadline-based algorithms, namely EDF and D^{over}.

2.3.4 GPS Based Approaches

In the context of assuring response time workflows, GPS was used in the scenarios where SLAs stipulated the mean response time, workflows are implemented strictly using sequence pattern, arrival rates and services time are exponentially distributed and component services are shared between workflows [141]. Addressing more complex scenarios, for example, where SLAs describe maximum response time, or where workflows can contain splits or loops, adds significant complexity to the problem. Another major drawback is the presumption of the stationary traffic, which is quite unlikely in certain scenarios [133].

2.4 Summary and Discussion of the Related Work

Research in the domain of scheduling began about fifty years ago resulting in a tremendous number of methods. Unfortunately, only a handful of problems appears to have polynomial time solutions.

In the domain of real-time systems, which directly addresses the issue of response time timeliness, most of the work has been done in the context of periodic systems allowing preemption. Even though service-oriented systems of this type exist (mostly used for industrial automation [130]), they do not represent a general case, as in the general case services do not support preemption. Nonetheless, attempts like [47] (no locks) showed that General Purpose Operating Systems can be used for implementing services supporting preemption. Even though, preemptive scheduling in such a case is limited to Static Priority, recent advancements in some open source projects such as Linux [53] make this direction a promising area. For example, the work in [37] demonstrated the possibility of performing preemptive EDF scheduling, yet only in the context of periodic jobs.

With regard to the other scheduling approaches, most studies have been conducted using simulations with idealistic server models, for example executing one job at a time. Unfortunately, the real-world services usually process several requests simultaneously and have more complex behaviour. Among the other assumptions that may not hold true is the assumption that job sizes are deterministic. Only a few studies focused on the scenarios where job sizes are stochastic [91, 139]. Therefore, the simulations, used in this thesis, were given a special focus in order to ensure that the simulation shows the results close to the ones which could have been obtained from experiments with real-world services.

In the context of the Service Oriented Architecture there are very few studies directly addressing the issue of enforcing Service Level Agreements by means of scheduling. Moreover, the problem of assuring SLAs for composite services using scheduling has gained very little attention. A set of rewarding algorithms addresses a similar problem, as does a study by Kao et al. [72]. However, the authors assume the composite services are either composed using a sequence pattern or using static precedence relations.

GPS scheduling suggests sharing service capacities among classes of clients. Consequently, in order to ensure the timeliness of the response time of a specific class, the appropriate provisioning is required [94]. Accordingly, it is possible to ensure the mean response time for service workflows for each class of composite service by employing GPS [141]. The main shortcomings of such approaches consist of a slow response to traffic parameter changes, as the traffic needs to be constantly monitored, as well as a failure to enforce timeliness of specific requests, as those ensure only the mean value of the response time, or the percentile of the response time.

Chapter 3
Proposed Solution

In this chapter the proposed solution is presented and an architecture enabling the proposed solution is outlined. The solution suggests ensuring SLAs for service workflows by means of bi-level scheduling. Bi-level scheduling is based upon the idea of first performing global scheduling and then conducting local scheduling. Global scheduling is responsible for translating workflows' SLA into deadlines for component service requests using a global scheduling policy. Local scheduling is performed on the proxy site and it is used for enforcing the timeliness requirements associated with the component service requests.

The first section contains the description of an architecture, as well as the details of local scheduling implementation using a proxy. The second section encompasses the proposed global scheduling policies, while the proposed local scheduling polices are presented in Section 3.3. Finally, the discussion of the proposed solution and analysis of its properties are presented in Section 3.4.

3.1 Proposed Architecture

The proposed architecture suggests dividing components responsible for performing scheduling into two groups: components residing on the workflow site [Figure 3.1] - implementing global scheduling - and the components located on the proxy site - implementing local scheduling.

The architecture can be employed with SOAP Web Services and composite services defined using workflow languages such as BPEL4WS [5]. Nonetheless, the same architecture can be used with other types of services, such as REST-full services [55]. Besides that, the solution is still valid in the scenarios where composite services are implemented using general

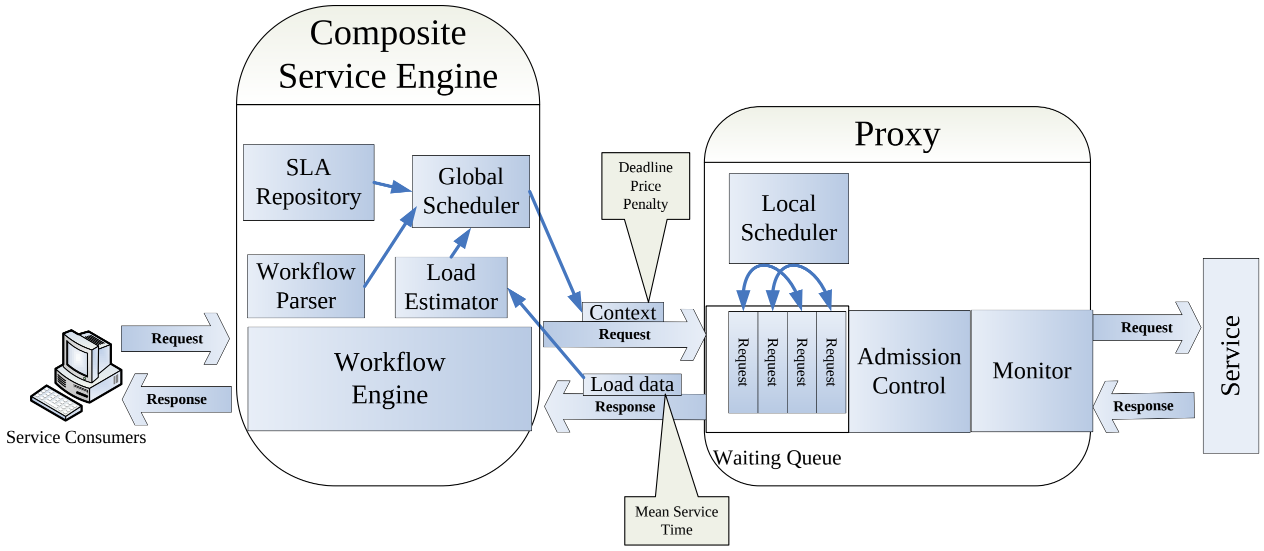

Figure 3.1: Proposed architecture for implementing bi-level scheduling.

purpose programming languages such as Java or C#. However, in such a case identifying the patterns of component service invocations [Section 1.1.4] and task dependencies is a cumbersome and daunting task.

The components implementing the global scheduling do not require substantial resources and thus can be hosted on the same machine as the composite service engine. At the same time, scheduling on the proxy site can be quite CPU intensive, and since the time of the scheduling decisions directly affects the response time of the requests, it is suggested to allocate dedicated physical servers for hosting the proxies.

3.1.1 Proxy Site Components

The components used for implementing local scheduling are admission control, local scheduler (scheduler) and monitor. Deployment of the component services scheduling requires control over the underlying message traffic. A typical way of performing this task is hiding the actual services behind reverse proxies [44, 49]. A proxy imposes the minimal overhead and thus a number of proxies can be hosted on the same physical machine. For example, HAProxy which is a proxy used for implementing load-balancing can achieve a throughput of almost 10 Gbps on a standard machine with Dual Core Intel CPU running at 2.66 Ghz[1].

[1]`http://haproxy.1wt.eu/10g.html`

There is a maximum number of requests that each service can process concurrently without thrashing - the MPL. The MPL can be found in an off-line manner, e.g. by performing stress load testing [Section 4.2]. As an alternative, the proxy can monitor the response time/throughput from the service and then use a controller to find the MPL [Section 2.2.1]. Admission control is responsible for ensuring that the maximum number of requests allowed in service simulataneously does not excceed the value of the MPL.

If an arriving request sees that the service is busy, i.e. the number of requests currently being processed is equal to the capacity of the service, the request is parked in a waiting queue. As soon as the service becomes available for handling another request, the queue is polled for the next request, which is sent to the service. The scheduler is responsible for implementing the scheduling discipline which determines the order of the requests in the waiting queue.

A monitor is a component responsible for recording the response times from the service and later providing this information to the load estimator.

3.1.2 Scheduling Proxy Implementation

The scheduling proxy is placed in front of a service in order to control the number of requests allowed in service, as well as to reorder the requests waiting for admission. In order to be deployed in a real-world setting, the scheduling proxy should meet all the following requirements:

1. Minimal latency overhead: A scheduling proxy should impose the minimal overhead both in terms of the computational complexity and request relaying overhead, as it directly impacts the quality of the scheduling decisions. In Section 4.1 it is shown that all of the suggested local scheduling policies require less than 1 msec for performing scheduling. Consequently, the proxy has to exhibit the minimal overhead from the message relaying perspective.

2. Scalability: A scheduling proxy should be able to correctly handle a potentially large number of concurrent connections as it is responsible for protecting a service during

load surges. At the same time, the number of concurrent connections should not affect performance of the proxy.

3. Independence from specific transport protocol: The scheduling proxy should operate on the transport level in order to enable scheduling SOAP and REST-full services[2].

Due to the reasons listed above, Erlang - a functional actor based programming language - has been chosen for implementing a scheduling proxy. Erlang has already proven its efficiency in building large scale concurrent systems [9], as it is specifically designed for running a large number of processes [10]. Furthermore, this programming language is ideal for this case as the proxy handles each incoming request in a separate process and thus the proxy needs to employ a large number of handling processes.

Erlang specifically targets the issue of minimizing process switching overhead by employing lightweight processes. Those processes have significantly lower context switching overhead as opposed to managed threads employed by languages such as Java or C#[3]. The main drawback of the thread-based architecture is that a proxy implemented using such a technique becomes susceptible to severe performance degradation when the number of threads reaches high values. Thus, employing such a threading model, when implementing a proxy, would make the proxy vulnerable to thrashing itself. In fact, in my work on QoS optimization for Web Services [45], a scheduling proxy implemented in Java was shown to fail at extreme loads. It is important to note that it is still possible to use threads for this problem. Yet it would require using an event-driven architecture [137] and asynchronous non-blocking IO libraries, e.g. Grizzly [4] or performing a number of tweaks on the compiler level [135]. For example, HAProxy - a load balancer, implemented in C according to an event-driven architecture, achieves throughput comparable to enterprise class load balancers.

A flow diagram illustrating functionality of the proxy can be found in Figure 3.2. First, an incoming socket connection is being assigned to a servant process (step 1), which is responsible

[2]Even though other protocols can be used for delivering service requests e.g. JMS, most of Web Services rely on HTTP for performing this task.

[3]Managed thread are often mapped one-to-one to native processes/threads. Other thread mapping models are also possible, however in practice one-to-one is the most popular (`http://download.oracle.com/javase/1,5.0/docs/relnotes/features.html`).

[4]GlasshFish, Grizzly, `http://grizzly.java.net/`

for communicating with a service client. After the connection has been established, an HTTP request is being sent by the client and received by the servant process. Upon the request arrival, a header containing information (scheduling context) used by a scheduling mechanism is extracted from the "scheduling" HTTP header [Figure 3.3]. In step 2, the servant notifies the scheduler process regarding the presence of the request, and it sends the request's scheduling context. After that, the scheduler process decides when the request should be forwarded to the service and then notifies the servant process that it has been allowed to proceed with the request execution. In case the service is busy, the request is considered to be "enqueued" and no response from the scheduler is sent to the servant. After the decision to dequeue has been made by the scheduler process, the scheduler will notify the servant with a message triggering the continuation of the request processing (step 3). If the request is not required to be placed in the queue, i.e. the number of requests being in service is less than the MPL, such a notification is sent immediately. After the arrival of the approving message the servant sends a message containing the request itself to a service invoker process (step 4). The invoker process forwards the request to the service and waits for the response (step 5). After the response has arrived (step 6), the monitor is being sent a message containing the response time, as well as the class of the client or client ID (step 7). The monitor records the data from the message in a database. Note that the service invoker processes are following a pool pattern, meaning that they do not die after completion of steps 5 and 6, but are returned to a pool for recycling. Besides that, each service invoker process has an already established socket connection with the service, thus service invokers avoid introducing additional latency when forwarding a request to the service. After the response has arrived to the service invoker (step 8), it is being forwarded to the servant which then forwards it back to the client (step 9).

3.1.3 Workflow Site Components

The proxy is responsible for the local scheduling part and admission control, while the global scheduling is performed on a workflow site[5]. Components residing within a workflow engine

[5]ActiveBPEL - `http://www.activevos.com/community-open-source.php` or Apache ODE (Orchestration Director Engine) - `http://ode.apache.org`

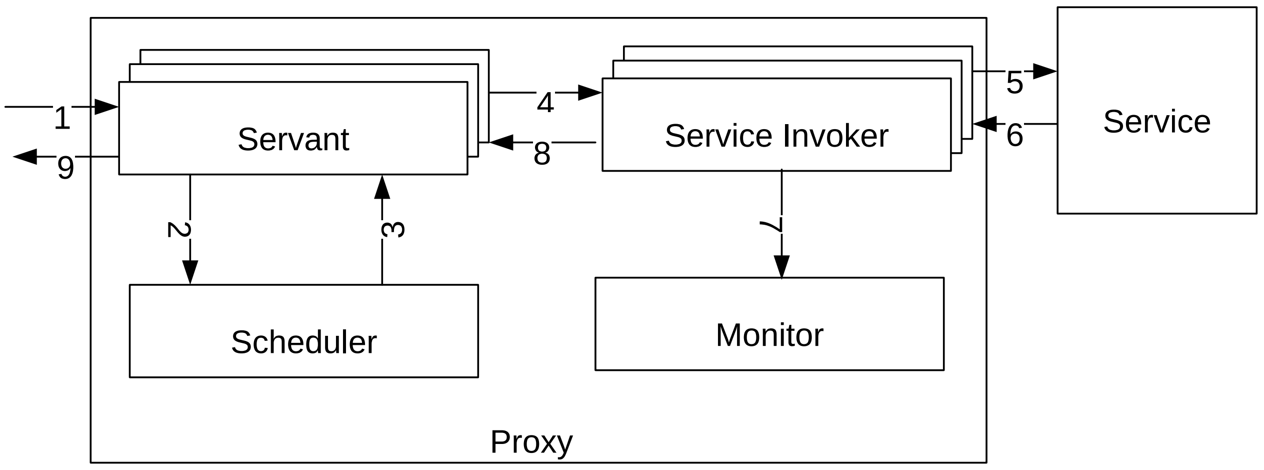

Figure 3.2: Scheduling proxy flow diagram.

```
POST /Services/ProjectStore HTTP/1.1
scheduling: [{deadline, 1.0}, {price, 1.0}, {penalty,  2.0}, {classname,
"class2"}]
SOAPAction: http://itracks.com/olfg/IProjectStoreService/SelectByProjectID
Connection: Keep-Alive
Host: 127.0.0.1:2010
Content-Type: text/xml; charset=utf-8
Content-Length: 273
User-Agent: Apache-HttpClient/4.0-beta2 (java 1.5)
Expect: 100-Continue

<s:Envelopexmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
<s:Body>
<SelectByProjectID xmlns="http://itracks.com/olfg">
<databaseServerName>Itracks-DB</databaseServerName>
<databaseName>OLFG</databaseName>
<projectID>1</projectID>
</SelectByProjectID>
</s:Body>
</s:Envelope>
```

Figure 3.3: An example of the HTTP request containing scheduling context encoded in the HTTP header “scheduling”.

combine load data with SLAs and produce a scheduling context [Figures 3.1 and 3.3]. In order to enable the global scheduling an existing workflow engine should be augmented with the following components:

- Workflow parser: The workflow parser is responsible for providing task dependency graphs to other components. Certain workflow languages, e.g. BPEL4WS use XML-structures to encode the control and data flow. Figure 3.4 contains an example of the code implementing parallel split pattern where services "StarLoan" and "unitedLoad" are invoked[6]. It is evident that parsing such a code and extracting all the dependencies between component service invocations is not a task as challenging as reverse engineering legacy Java or C# code.

- SLA repository: The SLA repository contains SLAs in a machine readable format, e.g. WS-Agreement, as well as the authentication means for locating an appropriate SLA for each incoming composite request.

- Load estimator: The load estimator periodically polls each proxy in order to obtain the information on the current response times from the services. In case the response time for each request is required, a piggyback method can be employed. The piggyback approach implies embedding the processing time in the header of every response coming back from the proxy to the workflow engine.

- Global scheduler: The global scheduler uses a task dependencies graph obtained from the workflow parser and mean processing times retrieved from the load estimator in order to find a deadline d and other parameters for each component service invocation. This information is embedded in an HTTP header and will be used later by a scheduling proxy [Figure 3.3].

  Any changes in the component services workload are automatically reflected on the mean response time. Thus, since the proxies are polled regularly, the mean response time of the component services in its turn affects the scheduling decisions. As a result,

[6] Oracle BPEL Process Manager Developer's Guide, Parallel Flow - `http://download.oracle.com/docs/cd/B31017_01/integrate.1013/b28981/parallel.htm`

the policy becomes adaptive and capable of operating even in the presence of sudden changes in component services loads.

3.2 Global Scheduling Policies

The global scheduler is governed by the specific scheduling policy which will be referred to as a global scheduling policy [Figure 1.7]. A global scheduling policy represents a policy which provides execution directives to local schedulers in order to assure that each workflow will meet its execution time constraints. More precisely, global scheduling policies are combinations of rules describing the process of decomposing a workflow's (global) deadline into deadlines for component requests.

In this thesis, the Ultimate Deadline, Equal Fraction and Proportional Fraction policies are considered. Ultimate Deadline assigns the workflow deadline (global deadline) to each component service invocation. Equal Fraction distributes the time allowed for a workflows's execution equally among all the participants of an orchestration. The Proportional Fraction policy augments Equal Fraction by also considering the average response time when calculating the deadlines for the component services.

Before the full description of the global scheduling policies is presented a process of segmentation of a workflow should be introduced. A worklflow segment Ω, or simply a segment, is a subset of a worklflow which receives one "message" on the input and produces exactly one "message" on the output. It can be a sequence of segments, segments invoked in parallel, exclusive choice segment, loop segment or an atomic service invocation. Each segment represents exactly one pattern. Therefore, each workflow can be viewed as a hierarchical composition of segments, e.g. a sequence of splits, a split of sequences, etc.

Segments are responsible for invoking segments in a sequence are called sequence segments. In case a segment contains invocations of segments in parallel, it is referred to as a split & synchronization segment. An atomic segment term will be used when a segment consists of a single invocation of a component service. Accordingly, exclusive choice segment is a segment consisting of segments, out of which only one will be invoked. Finally, a loop segment is a segment with a single subsegment invoked repeatedly. Ω_x will denote a workflow

```
...
<flow name="flow-1">
   <sequence>
          <scope name="UnitedLoan">
             <sequence>
                  <invoke name="invoke-2" partnerLink="unitedLoan"
                   portType="services:LoanService" operation="initiate"
                   inputVariable="loanApplication"/>
                   <receive createInstance="no" name="receive-1"
                    partnerLink="unitedLoan"
                    portType="services:LoanServiceCallback"
                    operation="onResult" variable="loanOffer1"/>
                    </sequence>
                    </scope>
             </sequence>
   <sequence>
          <scope name="StarLoan">
             <sequence>
                <invoke name="invoke-1" partnerLink="StarLoan"
                portType="services:LoanService" operation="initiate"
                inputVariable="loanApplication"/>
                  <pick name="pick-1">
                  </pick>
               </sequence>
          </scope>
    </sequence>
</flow>
...
```

Figure 3.4: An example of the BPEL code performing two parallel invocations of services "StarLoan" and "unitedLoad".

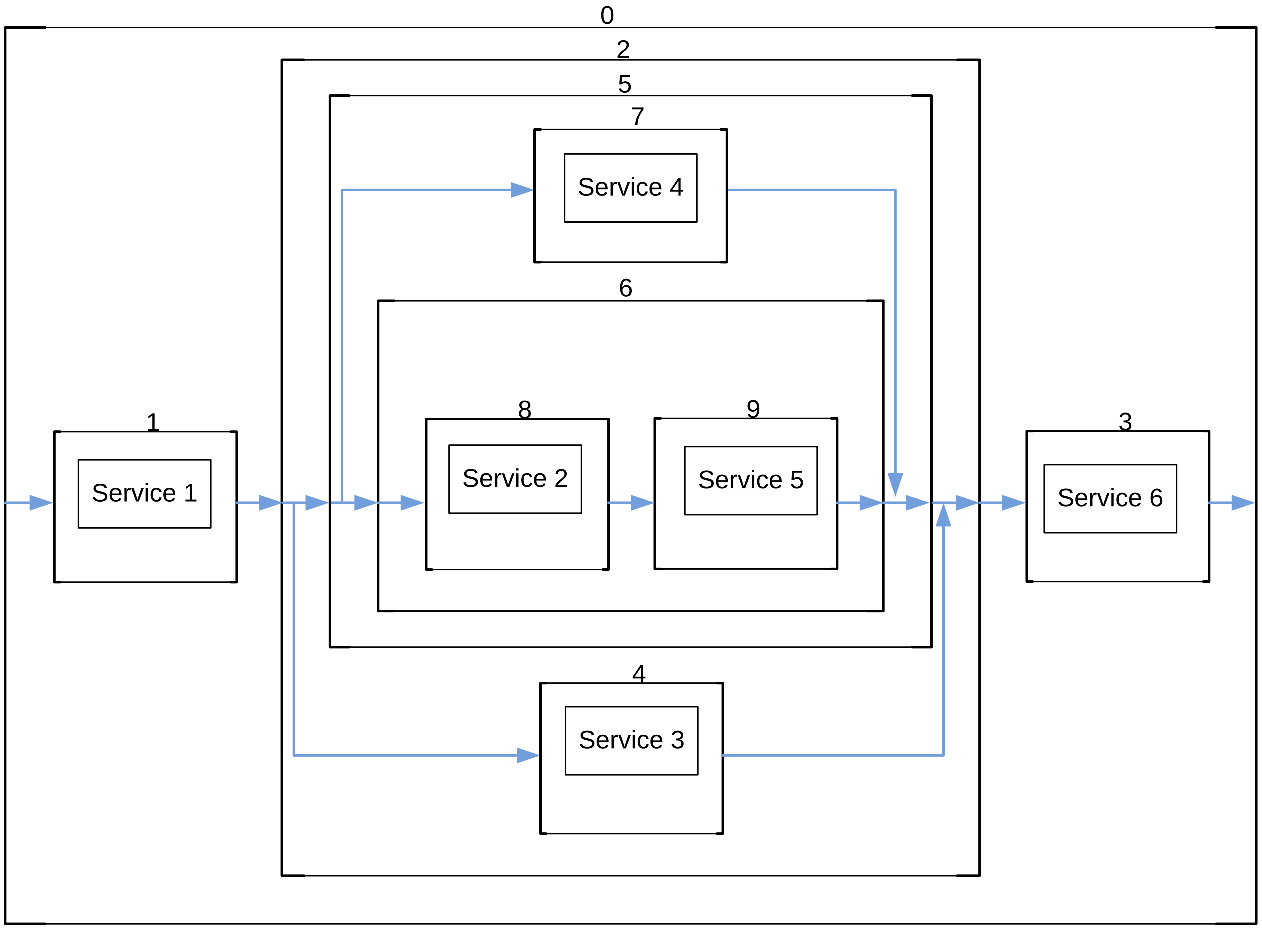

Figure 3.5: An example of a composite service segmentation.

segment x and all the subsegment components j in segment x will be marked as $\Omega_{x,j}$.

An example of the segmentation is shown in Figure 3.5. The summary of the interrelations between the segments is presented in Table 3.1. As it can be seen, the whole workflow is viewed as a sequence segment 0 which contains segments 1,2 and 3. Segments 1 and 3 are atomic segments, while segment 2 is a parallel split segment. Segment 2 contains two branches, segment 4 – atomic segment and segment 5. In its turn segment 5 is also a parallel split segment which has two branches – an atomic segment 7 and a sequence segment 6. Finally, segment 6 is a sequential invocation of two atomic segments 8 and 9.

3.2.1 Ultimate Deadline (UD)

Ultimate Deadline (UD) has the minimal set of requirements in order to be deployed. The only information required for deploying this policy is the deadline d_i assigned to the workflow

Table 3.1: A table explaining segmentation presented in Figure 3.5.

Segment #	Type	Subsegments
0	sequence	1,2,3
1	atomic	
2	split&synchronization	4,5
3	atomic	
4	atomic	
5	split&synchronization	6,7
6	sequence	8,9
7	atomic	
8	atomic	
9	atomic	

Table 3.2: Global scheduling policies deployment requirements.

	Deadlines	Workflow Structure	Mean Processing Times
Ultimate Deadline (UD)	Yes	-	-
Equal Fraction (EF)	Yes	Yes	-
Proportional Fraction (PF)	Yes	Yes	Yes

i. UD operates by passing the same deadline d_i to all of its component service invocations.

The assignment of the deadlines is performed recursively according to Algorithm 1. The deadlines are assigned by invoking procedure UD(Ω_{main},D), where Ω_{main} is the main segment of a workflow and D is the worfklow's relative deadline, extracted from an SLA.

3.2.2 Equal Fraction (EF)

Equal Fraction is a global scheduling policy which assigns an equal fraction of the time before the deadline to each component service invocation in the slowest branch of a workflow. In case of a sequence of atomic service invocations the process of finding those fractions is rather trivial [72]. However, in case a segment contains other segments, the situation is more complicated. Therefore, a special method has been developed to address the issue.

Algorithm 1 An algorithm for assigning deadlines according to UD global scheduling policy.

```
Require: $\Omega_x$ is a workflow segment
Require: $D$ is a relative deadline
Require: $r$ is a composite request arrival time
  procedure UD($r$,$\Omega_x$,$D$)
  if $\Omega_x$ is atomic segment then
    $d_{\Omega_x}$:= $r$+$D$
  else
    for all $\Omega_{x,m} \in \Omega_x$ do
      UD($r$, $\Omega_{i,m}$, $D$)
    end for
  end if
```

Before the method is presented, a procedure which finds the longest path of component service invocations should be introduced. Often such a path is referred to as a Critical Path [112]. In case of static workflows, the Critical Path can be found using the Critical Path Method (CPM). However, the CPM cannot be applied in the scenarios where the workflows contain dynamic components. Thus, a special recursive procedure for performing this task was developed [Alg. 2]. The procedure is based upon the idea of using a pessimistic approximation of the dynamic components together with the static ones. Loop segments are approximated with sequence segments, and exclusive choice segments are approximated with parallel split segments. For example, a loop which repeats no more than 10 times can be approximated by a sequence containing 10 elements. An illustration of calculation of the Critical Path lengths for each segment in the workflow is presented in Figure 3.6. The number in the lower right corner of each segment is the Critical Path length of the segment.

The length of the Critical Path represents the number of elements in the longest chain of component service invocations which are to be executed in the workflow. EF suggests equally dividing the deadline between the components in the Critical Path. An illustration of such deadline allocation is presented in Figure 3.6. The number in the lower corner is the relative deadline allocated to the subsegment. The workflow has to be completed in 1.0 sec. Atomic segments (component service invocations) 1, 8, 9 and 3 constitute the Critical Path,

Algorithm 2 An algorithm for finding a number of component service invocations in the Critical Path.

```
Require: $\Omega_x$ is a workflow segment
  procedure CRITICAL_PATH_LENGTH($\Omega_x$)
  switch $\Omega_x$ is
  case atomic segment:
      return 1
  end case
  case sequence segment:
      sum:= 0
      for all $\Omega_{x,m} \in \Omega_x$ do
          sum:= sum + CRITICAL_PATH_LENGTH($\Omega_{x,m}$)
      end for
      return sum
  end case
  case loop segment:
      return the maximum number of iterations*CRITICAL_PATH_LENGTH($\Omega_{x,m}$) {
  $\Omega_{x,m}$ is a subsegment of $\Omega_x$}
  end case
  case exclusive choice segment or parallel split:
      max:= -1
      for all $\Omega_{x,m} \in \Omega_x$
          if max $\leq$ CRITICAL_PATH_LENGTH($\Omega_{x,m}$) then
              max:= CRITICAL_PATH_LENGTH($\Omega_{x,m}$)
          end if
      end for
      return max
  end case
  end switch
```

Algorithm 3 An algorithm for assigning deadlines according to EF global scheduling policy.

```
Require: $\Omega_x$ is a workflow segment
Require: $D$ is a relative deadline
Require: $r$ is a request arrival time
  procedure EF($r$, $\Omega_x$, $D$)
  switch $\Omega_x$ is
  case atomic segment:
      $d_{\Omega_x}$:= $r$+$D$
  end case
  case sequence segment:
      $\delta$:= $D$ / CRITICAL_PATH_LENGTH($\Omega_x$)
      q:= $r$
      for all $\Omega_{x,m} \in \Omega_x$ do
          EF(q, $\Omega_{x,m}$, q+$\delta$)
          q:= q+ $\delta$
      end for
  end case
  case loop segment:
      EF($r$, $\Omega_{x,m}$, $D$/the maximum number of iterations) { $\Omega_{x,m}$ is a subsegment of $\Omega_x$}
  end case
  case exclusive choice segment or parallel split:
      for all $\Omega_{x,m} \in \Omega_x$ do
          EF($r$, $\Omega_{x,m}$, $D$)
      end for
  end case
  end switch
```

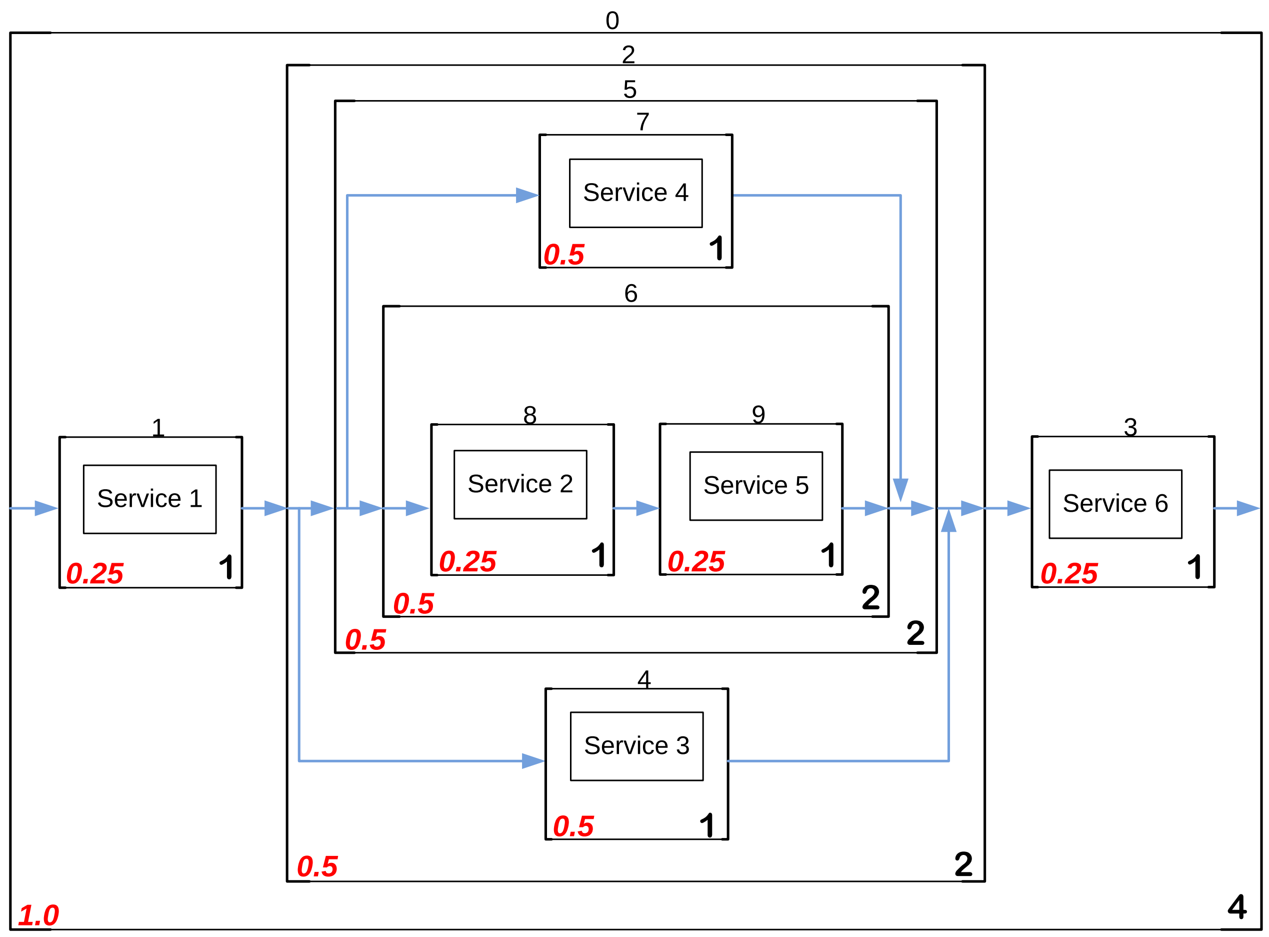

Figure 3.6: An example of EF applied to a composite service.

thus the deadline is being divided equally among those four atomic segments. As a result, all these segments are being assigned relative deadlines of 0.25 sec. Since segment 6 is a sequence of segments 8 and 9, it will be processed in 0.5 sec (assuming that the deadlines are met). Accordingly, segment 6 is allocated 0.5 sec for its processing. Segment 7 is executed simultaneously with segment 6, therefore processing it in less than 0.5 sec would be irrational, as its result is synchronized to the result of segment 6. Thus, it is also assigned a relative deadline of 0.5 sec. Accordingly, the deadline of segment 5 is 0.5 sec. Segments 4 and 5 are parallel branches, therefore their deadlines must be equal, consequently segment 4 is being assigned a relative deadline of 0.5 sec. In such a way EF prioritizes the execution of critical component services determining the completion time of a workflow, meanwhile less critical component service invocations are allocated larger deadlines. In this way, EF releases non-critical resources of the workflow to worflows for which those resources may be critical.

The complete and precise description of the process of the deadlines assignment is presented in Algorithm 3. As the segmentation is recursive, the calculation of the deadlines is recursive too. For the top most segment which represents the workflow itself, r is set to the arrival time of the composite service request, while D is found from the corresponding SLA. Using the process described in Algorithm 3, the deadlines and arrival times are found for each subsegment. The process is then repeated with each subsegment of each subsegment and so on. The calculation is completed when the deadlines and arrival times are assigned to every component service invocation used in the workflow. The process of assigning the deadlines is initiated by invoking EF(r, Ω_{main}, D), where r is the arrival time of the composite service request, Ω_{main} is the main segment and D is the relative deadline of the workflow.

3.2.3 Proportional Fraction (PF)

In contrast to EF, the Proportional Fraction (PF) policy recognizes that certain component requests may require more time to be processed than the others. Thus, it assigns deadlines according to an average response time of each component service invocation, which is determined by the load the component services are facing. Therefore, if a certain service is facing a higher load, its response time increases, resulting in allocating more time for executing a request which may take longer to process.

The algorithm for assigning the deadlines to component service invocations employs a procedure called MEAN_RESPONSE_TIME(Ω_x). The procedure returns a segment x average response time calculated using the sliding window of the length of 30 minutes. An example of this procedure applied to a worklow is presented in Figure 3.7. The number in the right lower corner of each segment is the mean response time of the segment. The response time of segment 2 is determined by its slowest branch as it is a split segment. The slowest branch is segment 5, which requires 0.5 sec on average, thus executing segment 2 also takes 0.5 sec. It is evident, that atomic segments 1,8,9 and 3 form the Critical Path as this is the slowest chain of the component requests. Note, that the PF policy uses the procedure called NORMALIZED_RESPONSE_TIME which returns the response time of a segment divided by the workflow completion time.

PF policy acts in a way similar to EF, except that atomic segments now have different "weights" - average response times. An illustration of the algorithm's functionality is shown in Figure 3.7. The number in the lower left corner of each segment is the deadline assigned to the segment. The workflow's deadline is 1.0. This deadline is divided along the Critical Path proportionally to the response times of the component service invocations. As a result, atomic segment 1 has to be completed in 0.1 sec, atomic segment 8 in 0.1, atomic segment 9 in 0.4 and atomic segment 3 in 0.4 sec. Since, Segment 6 is executed in parallel to segment 7 and the projected completion time (determined by its deadline) of segment 6 is 0.5 sec, segment 7 is also given a deadline of 0.5 sec. The same applies to segments 5 and 4.

PF enforces the Critical Path to be executed in time to ensure that the workflow's deadline is met. Since service 5 is slower than the others, possibly to higher load or longer service times, it is given a larger deadline. By doing so, PF attempts to reduce the stress on service 5. Similar to EF, PF releases non-critical component services to other workflows, for which those service may be more critical. Besides that, it also recognizes the potential bottlenecks in worfklows and adjusts the deadlines to ensure that those bottlenecks are allocated sufficient amounts of time.

An algorithm for assigning the deadlines according to the PF policy can be found in Algorithm 5. The process is initiated by invoking PF(r, Ω_{main}, D), where r is the arrival time of the composite service request. Ω_{main} is the main segment and D is the relative deadline of the workflow.

Algorithm 4 An algorithm for finding normalized response time of a segment.

```
Require: $\Omega_x$ is a workflow segment
  procedure NORMALIZED_RESPONSE_TIME($\Omega_x$)
  return MEAN_RESPONSE_TIME($\Omega_x$) / MEAN_RESPONSE_TIME($\Omega_{main}$) {$\Omega_{main}$ is the main segment of the workflow}
```

The average response time for each segment is obtained by maintaining response time from component services [Subsec. 3.1.1]. The algorithm requires a certain amount of time in order to be initialized. In order to minimize the losses during the boostrapping process, it is assumed that all services have equal response times. Thus, the initial deadline assignment is performed according to the EF policy. Later, as the historical time is being accumulated,

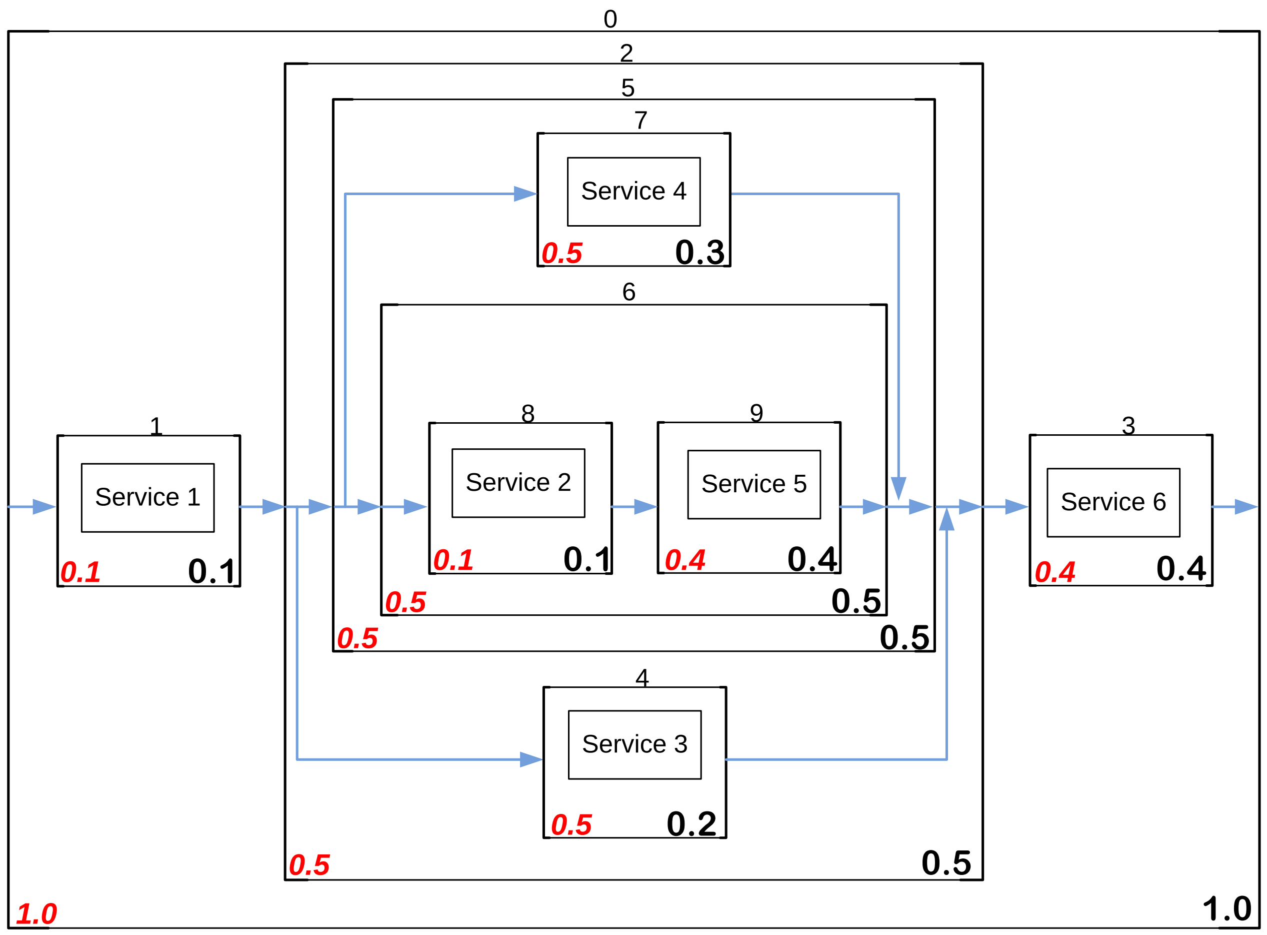

Figure 3.7: An example of PF applied to a composite service.

Algorithm 5 An algorithm for assigning deadlines according to PF global scheduling policy.

```
Require: $\Omega_x$ is a workflow segment
Require: $D$ is a relative deadline
Require: $r$ is a composite request arrival time
  procedure PF($r$,$\Omega_x$, $D$)
  switch $\Omega_x$ is
  case atomic segment:
      $d_{\Omega_x}$:= $r$+$D$
  end case
  case sequence segment:
      $\delta$:= $D$ / NORMALIZED_RESPONSE_TIME($\Omega_x$)
      q:= $r$
      for all $\Omega_{x,m} \in \Omega_x$ do
          PF(q, $\Omega_{x,m}$, q+$\delta$*NORMALIZED_RESPONSE_TIME($\Omega_{x,m}$)
          q:= q+ $\delta$*NORMALIZED_RESPONSE_TIME($\Omega_{x,m}$)
      end for
  end case
  case loop segment:
      PF($r$,$\Omega_{x,m}$, $D$) { $\Omega_{x,m}$ is a subsegment of $\Omega_x$}
  end case
  case exclusive choice segment or parallel split:
      for all $\Omega_{x,m} \in \Omega_x$ do
          PF($r$, $\Omega_{x,m}$, $D$)
      end for
  end case
  end switch
```

the deadlines are adjusted accordingly.

3.3 Local Scheduling Policies

Local scheduling policies are responsible for ordering the requests arriving to component services. Such scheduling can be implemented on the proxy site or be a part of an ESB [Section 1.1.5]. Four different local scheduling policies are suggested, Highest Value First, Highest Ratio First, Earliest Deadline First and the algorithm proposed by Lawler in [81].

1. Highest Value First (HVF) : HVF is a Static Priority policy, where the priority of the request is determined by its value v. This policy suggests prioritizing the component requests strictly according to the penalty (business value) associated with a failure. Thus, the more expensive requests have better chances to meet their deadlines. Note that in a real-world setting there is a strong correlation between the value and penalty, as the higher the price the higher the penalty.

   In case the queue is implemented as a balanced tree, the enqueue operation has the $O(log(N))$ complexity, where N is the number of elements in the queue. While, the dequeue operation has the complexity of $O(1)$, as it is essentially the removal of the maximum element.

2. Earliest Deadline First (EDF): EDF serves incoming requests according to the order of their deadlines d, from sooner to later [118]. In case EDF queue is implemented as a balanced tree where the requests are ordered according to d, the complexity of the enqueue and dequeue operations will be $O(log(N))$ and $O(1)$ respectively.

3. Highest Ratio First (HRF):

   HRF is a dynamic priority policy, where the dynamic priority is calculated according to the following formula:

   $q(t) = \begin{cases} (t-d)/v & t < d \\ (t-d) * v & t \geq d \end{cases}$, where t is the current time, and q is the priority. The requests are ordered in the decreasing order of q. This policy combines both EDF and

HVF [44, 46]. The main idea behind this policy is that the request that is processed ahead of the others either has a closer deadline or a larger business value. This policy aims to overcome EDF's "domino" effect [93] (emerging in case of overloads), when requests are missing the deadline while still being processed. With HRF it is possible to mitigate the "domino" effect by giving higher priority only to the requests with the highest business values and the closest deadlines. Since HRF employs the dynamic priorities dependent on the time, every time the queue is polled for the request with the maximum priority, the priorities of all the requests need to be re-evaluated. Thus, the complexity of the dequeue operation is $O(n)$. As for the enqueue operation, it can be performed in $O(1)$ steps as the requests are not organized in any particular order.

4. Lawler's Scheduling Method (LSM): Lawler's Scheduling Method [81] guarantees the optimal solution for a single service if the information on request sizes is available and all the requests are available at time zero. However, in the case of constant request arrivals, the method cannot ensure that the sum of penalties is minimized. Denote the computing requirements (job sizes) for service requests by p_{ij}, the number of elements in a waiting queue j by n_j, and functions which return the penalty for completing a component request ij at time t by $f_{ij}(t)$. As for the penalization function, it is chosen depending on the type of SLA against which the response time of the workflow will be evaluated. Note that in [81] it is shown that the algorithm is optimal for an arbitrary non-decreasing penalty function.

   If we refer to the set of unscheduled requests as U_j and to the queue of scheduled invocations as $\pi_j(k)$, then the algorithm for scheduling the queue in front of the component service j can be described as the following:

```
$U_j := 1...n_j$
$p_j := \sum_{i=1}^{n_j} p_{ij}$
for $k := n_j$ DOWNTO 1 do
    Find job $x \in S$ such that $f_{xj}(p_j)$ is minimal
    $U_j = U_j \backslash \{x\}$
    $\pi_j(k) := x$
    $p_j := p_j - p_{xj}$
  **end for**
```

The enqueue complexity of this scheduling policy is $O(1)$ as it does not require reordering the requests, while the complexity of the dequeue operation is $O(N^2)$.

3.4 Discussion of Proposed Solution

The global scheduling policy informs the local schedulers about the timeliness and the importance of component service requests, giving them a degree of autonomy by allowing scheduling the component requests according to their own rules. In this way, the local and global schedulers become loosely coupled. Consequently, local policies can be easily interchanged and different local policies can be deployed at the same time with the same global policy. For instance, for some services job sizes cannot be accurately predicted and LSM cannot be used.

The presented scheduling policies differ not only in terms of the computational complexity, but also in terms of required information. HVF, EDF+UD, HRF+UD, LSM+UD are the simplest scheduling policies which need only data which is available from corresponding SLAs; consequently, they have the minimal set of requirements.

In order to use EDF+EF, HRF+EF and LSM+EF, it is necessary to have the complete information regarding the dependencies between component service invocations. In case composite languages are defined using workflow languages such as BPEL4WS, this can be as trivial as parsing an XML file. While in case composite services are written in Java, C# or any other general purpose programming language, this task is not as simple. Nonetheless, it can be done using two methods.

The first method would require human intervention in order to analyze the code of the composite service and to extract the required information manually. Such information can be extracted from the supporting documentation, such as sequence diagrams. Such an approach is quite error prone, as any change in the interaction between workflows and component services needs to be reflected in the documentation which often does not happen in a real-world setting. Relying on the outdated documentation will most likely result in incorrect scheduling decisions.

The second method is more general and relies on reverse engineering tools and profilers[7] in order to establish the dependence between component service invocations. The main shortcoming of such an approach consists of tight coupling with a specific programming language. Besides, unifying the data obtained from profilers for various programming languages is not a trivial task either.

In case of EDF+PF, HRF+PF and LSM+PF the data on the average response time from component services is needed as well. While LSM+* policies on top of that also require the a priori knowledge on the job sizes. Obtaining such information can be rather complicated task. However, the practice shows that for Web Services the quality of prediction can be quite high. For example, in [45] it was shown that the correlation coefficient between the expected service time and its actual value can be as high as 0.7. The summary of the deployment requirements is presented in Table 3.3.

Apart from finding the MPLs for component services and job sizes, none of the policies require human intervention in order to be used after the deployment. Moreover, the PF policy makes appropriate adjustments when the load changes. Therefore, the solution meets the adaptivity requirement [Section1.2, Item 2].

The proposed solution also works in the scenarios where workflows are hosted on multiple workflow engines. It is possible due to the fact that the global schedulers located on workflow engines communicate with scheduling proxies by means of embedding scheduling context in requests. As a consequence, the solution does not require a single point of control, i.e. it is decentralized [Section 1.2, Item 3].

The solution presented in this chapter suggests employing scheduling proxies for gaining control over the sequence of incoming requests in order to implement request scheduling. As a result such a solution allows deploying scheduling in a transparent manner, i.e. without modifying the code base of the services [Section 1.2, Item 4].

Finally, since scheduling is performed before the requests are being admitted in service, scheduling proxies do not have any specific requirements for services in terms of the underlying OS or language of implementation. Consequently, the proposed solution is also platform independent [Section 1.2, Item 5].

[7]For example, TPTP `http://www.eclipse.org/tptp/`

Table 3.3: Local scheduling policies deployment requirements.

Name	Workflow structure	Mean response time	Job sizes
HVF	-	-	-
EDF+UD	-	-	-
EDF+EF	YES	-	-
EDF+PF	YES	YES	-
HRF+UD	-	-	-
HRF+EF	YES	-	-
HRF+PF	YES	YES	-
LSM+UD	-	-	YES
LSM+EF	YES	-	YES
LSM+PF	YES	YES	YES

Chapter 4
Experiments

The main objective of the experiments is evaluating the efficiency of combinations of the proposed scheduling policies in various contexts. The advantages of performing scheduling on the proxy site are investigated by means of experiments with a commercial Web Service[1]. The data obtained from this experiment is then used for calibrating the model of a single service. Service workflows are modeled by means of combining several models of a single service and such a model is used for benchmarking the quality of the proposed scheduling algorithms.

This chapter is structured as follows. Section 4.1 contains the description of an experiment conducted to find the maximum computational complexity of local scheduling algorithms. The process of finding the MPL for a Web Service is presented in Section 4.2. Section 4.3 is comprised of the experiments with a real-world Web Service and a scheduling proxy implementing Static Priority scheduling. Section 4.4 contains the details of validating an event-driven simulator using the results of the experiment from Section 4.3. The evaluation of the local scheduling policies can be found in Section 4.5. Finally, the experiments used for analysing global scheduling policies are presented in Section 4.6.

4.1 Non-preemptive Scheduling Overhead Limitations

Service orientation emphasizes customization and is primarily designed for simplifying business-to-business interaction as well as implementing large scale distributed enterprise computing systems. Therefore, the granularity of service requests and workload scenarios impose SOA specific constraints on scheduling overhead, which in turn compels seeking SOA specific

[1]`http://itracks.com`

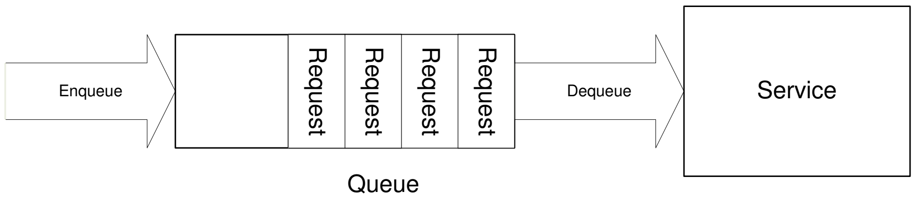

Figure 4.1: Queueing the requests.

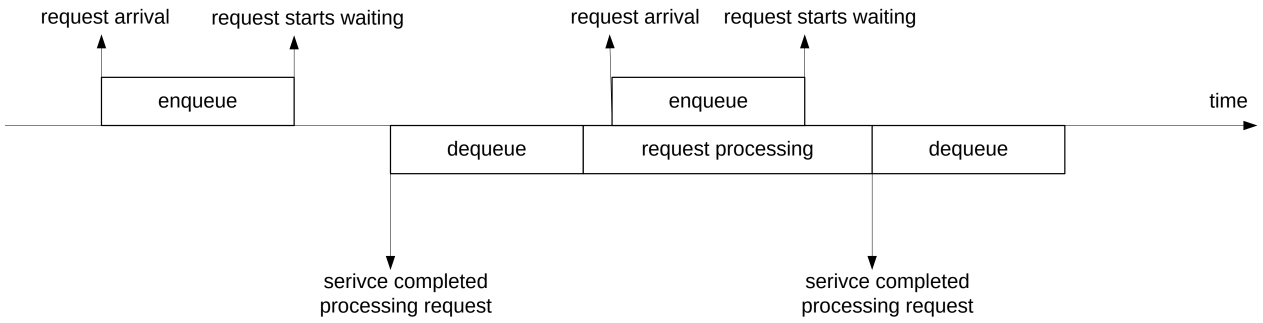

Figure 4.2: Scheduling overhead.

methods.

Service oriented systems typically have somewhat low response time and high throughput. For example, in works [45, 43] it has been shown that in the TPC-App benchmark [129], representing a typical commercial Web service, the processing times of service requests fall within a range of 10-500 milliseconds. It is important to understand that the overhead of more than a few milliseconds already represents a substantial obstacle for deploying a specific algorithm. Therefore, it is absolutely essential to evaluate how the complexity of a specific scheduling algorithm affects the latency added to the response time of requests.

Since scheduling is performed on the proxy site, each arriving request is being placed in a queue in case it cannot be accepted in service [Figure 4.1]. The complexity of the enqueue operation depends on the scheduling algorithm. For example, in the case of a First-In-First-Out policy, it can be as small as $O(1)$, while for the Static Priority policy, an implementation using a balanced tree has a complexity of $O(log(N))$. Since the algorithm is typically used repeatedly, the complexity of the dequeue operation has a significant impact on the response time. Besides that, the dequeue operation overhead is always added to the response time of the request. The enqueue operation, however, can be performed asynchronously and its overhead has a lesser impact on the response time. The process is illustrated in Figure 4.2. The enqueue operation can be performed while a request is being processed by a service, and

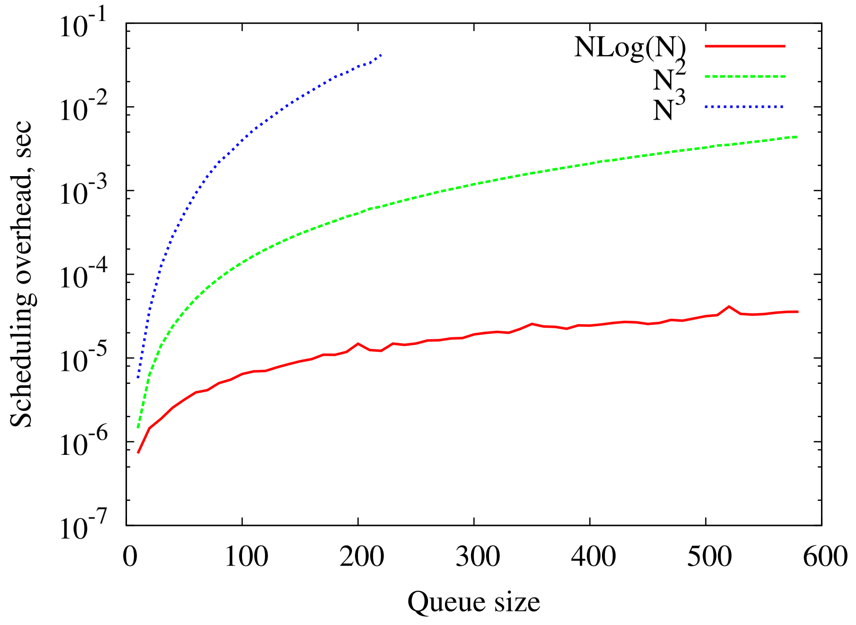

Figure 4.3: Scheduling overhead in terms of latency.

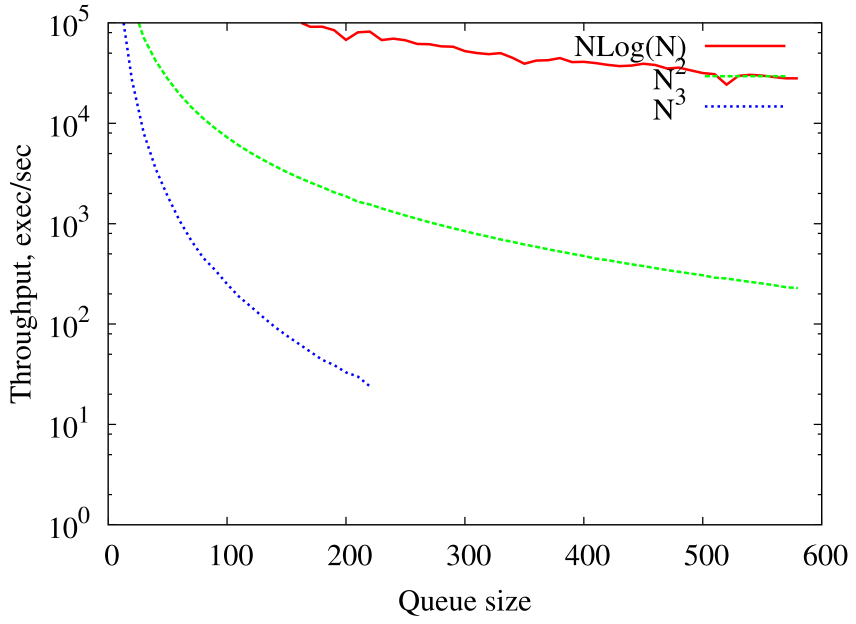

Figure 4.4: Throughput of various schedulers running on a single core.

the dequeue operation is always performed when the service becomes available for accepting the next request.

Figures 4.3 and 4.4 demonstrate the scheduling overhead (latency and throughput) dependence with respect to the complexity of a scheduling algorithm. The experiments were performed on a 3.2 Ghz Pentium 4 Xeon machine. Each algorithm used a single thread of execution; consequently, only one core was utilized. The algorithm simulated scheduling operations which require $Nlog(N)$, N^2 and N^3 operations, where N is the number of elements in a queue. Each operation also included a floating point division and a multiplication, which are the most common operations in most of the scheduling algorithms, e.g. HRF, LSM. The x axis represents the number of elements in a queue, while the y axis depicts the time required for making a scheduling decision. The chart in Figure 4.4 displays the maximum number of repetitions of a scheduling algorithm performed in one second, and thus represents the throughput of a scheduler utilizing a single core at 100%.

In order to generalize the results, it is assumed that the highest acceptable overhead is not greater than 1 millisecond. After comparing the results shown in Figure 4.3, one can see that the N^2 algorithm always demonstrates strong results while the overhead latency starts exceeding the threshold of 1 millisecond only if the queue grows beyond 500 elements. This makes N^2 scheduling methods (e.g. LSM) acceptable even though the scheduling would have to be performed on a separate CPU. This is still a likely option in the scenarios where an ESB is employed [Section 1.1.5]. Consequently, given the current state of the modern hardware, only the local scheduling algorithms with the complexity of $O(N^2)$ or better can be employed with service-oriented systems.

4.2 Finding the Multiprogramming Level

This section describes an experiment conducted in order to find the optimal number of jobs being concurrently processed in order to prevent thrashing and avoid subsequent throughput degradation, as well as to enable external scheduling. The experiment was performed in order to observe the effect of the maximum number of concurrently processed jobs on the performance of a typical commercial Web Service. The results of this experiment are used

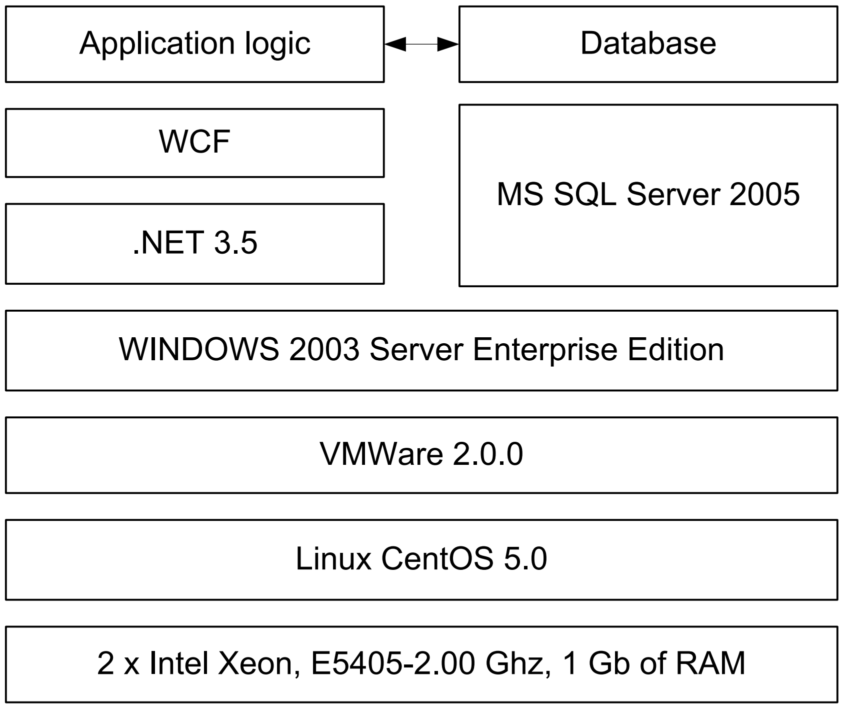

Figure 4.5: Web Service "X" software and hardware stack.

in the subsequent sections.

A Web Service "X" used internally by Itracks[2] was chosen as a subject of the study. The Web Service represents one of the services employed by the company's front-end web applications[3]. The service is built according to a two-tier architecture, where the first tier accommodates the business logic, and the second tier is a backend database. Both tiers were hosted on the same virtualized machine, running Windows 2003 Server (1 Intel Pentium Xeon E5405 CPU at 2.00 Ghz, 2 GB of RAM). VMWAre 2.0 was used as a virtualization platform and the Linux CentOS 5.0 operating system served as the host OS (Quad Core Intel Xeon E540, 16GB of RAM, 250GB 7.2K RPM Serial ATA 3Gbps Hard Drive). Windows Communication Framework (WCF) running on .NET CLR 3.5, was used as a middleware responsible for parsing incoming SOAP requests and generating responses conveyed via the HTTP protocol. A Microsoft SQL Server 2005 was used a backend database. The WCF was configured to accept all incoming requests, i.e. the internal admission control mechanisms

[2]Itracks (`http://itracks.com/`) is a mid-size company performing market research using web based applications.

[3]Unfortunately, the nature or the details of business logic of the employed Web Service cannot be discussed in great details due to a non-disclosure agreement.

were disabled. The physical host was not shared with any other virtual machines or processes except for the default system ones.

Each request was handled in a single thread, meaning that the processing of a request was bound to one core. The requests consisted of CPU operations, database queries and other I/O operations.

The client load generator was built in Erlang and implemented a closed model with zero think time. Thus, each client was issuing a new request immediately after obtaining a response. Note that when the think time is set to zero, the number of clients corresponds to the number of jobs submitted. As a result, the number of clients corresponded to the number of requests being processed simultaneously.

The load generator employed HTTP 1.1 with the persistent connections. Therefore, each client reused an established TCP connection, and as a result the overhead associated with a new socket opening was eliminated. HTTP pipelining [54] of requests was disabled due to WCF's inability to support it correctly.

The load generator was hosted on 2xIntel Xeon running at 3.2 Ghz, 9 Gb of RAM and 1 Gbps network connection. The location of the load generator was 5 networks hops away from the server hosting the service. The average packet round trip time was less then 2 ms and constituted less than 1% of the service time. The bandwidth of the network connection (1 Gbps) exceeded the highest generated traffic (5 Mbps) more than one hundred times. Thus, it was assumed that the network latency did not have any significant impact on the response time.

Data samples were collected every 10 seconds. Each run lasted for 120 seconds during which the number of clients stayed fixed. Each data point in the following charts [Figures 4.6, 4.7, 4.8, 4.9, 4.10, 4.11 and 4.12] apart from the mean value also display the 95% confidence intervals calculated using the Student's t-distribution.

As it can be seen from a chart in Figure 4.6, the throughput has a unimodal dependence with respect to the number of requests simultaneously admitted in the service. The service demonstrates the maximum throughput at the point where the MPL is set to 7. Further increases of the MPL results in the deterioration of the performance, mostly due to the increased lock contention [Figure 4.7], which directly affects lock waiting time [Figure 4.8]

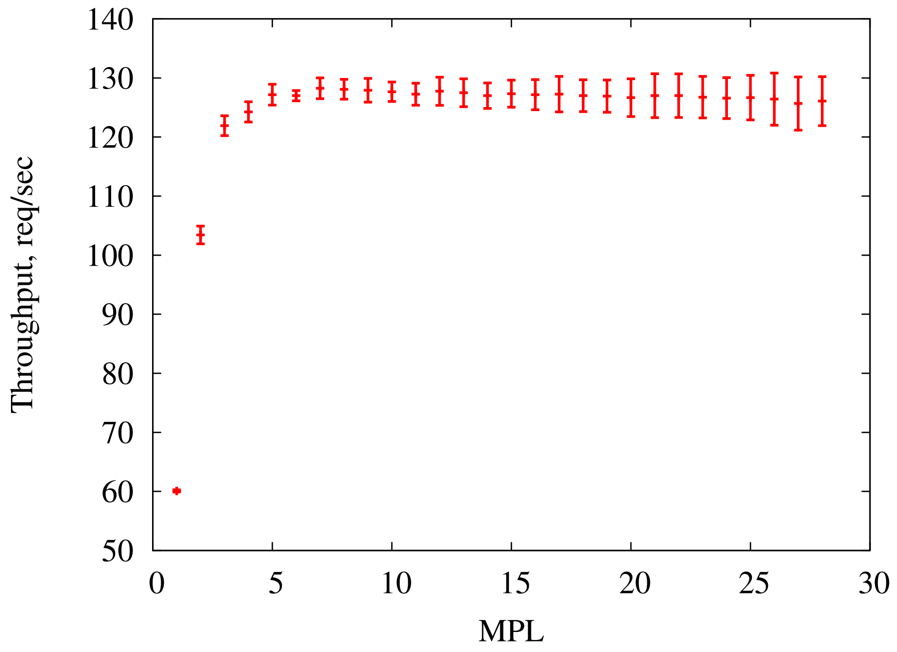

Figure 4.6: Web Service "X" throughput with respect to the MPL.

and the time spent for executing the business logic part of the request handling [Figure 4.9]. Note that the time spent on processing the business logic is not exactly the same as the service time, as it excludes the waiting time, as well as the time spent on decoding (parsing) the incoming request and encoding the response. The waiting time includes the time waiting for the CPU, I/O and locks. Since the service time refers to the time since the request has been allowed in service till the response obtained, in this case the service time is equal to the response time as there is no external waiting queue. Another important observation is that the actual number of threads running in parallel when processing the incoming requests is lower than the number of requests allowed in the service. Such a behaviour emerges when the CPU utilization reaches its maximum value [Figure 4.10]. In this case, all requests are first handled by an acceptor thread which accepts the incoming request and then delegates it to a handling thread. One possible explanation of this phenomena is that if the CPU utilization is very high and all the threads are sharing the CPU in an equal manner, the acceptor thread suffers from getting less CPU time. As a result, the rate at which the requests are accepted is lower and consequently the actual number of the handling threads is less than the number of requests allowed in the service [Figure 4.11].

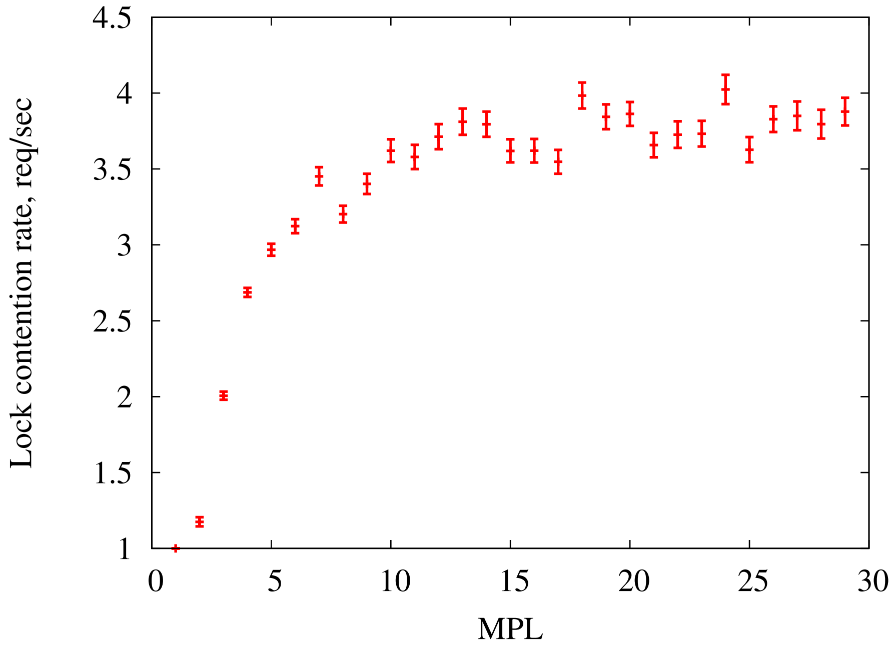

Figure 4.7: Web Service "X" mean lock contention rate with respect to the MPL.

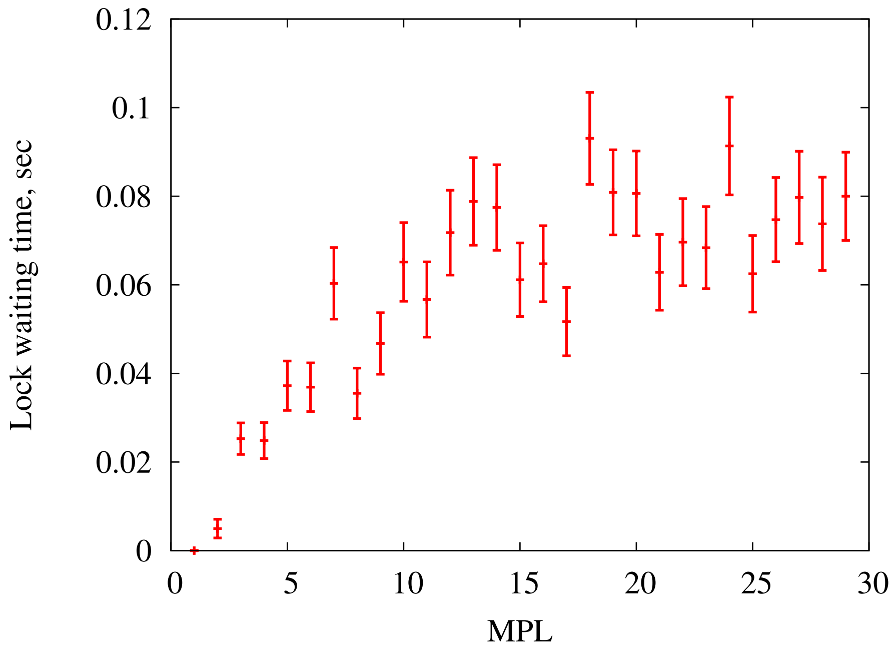

Figure 4.8: Web Service "X" mean lock waiting time with respect to the MPL.

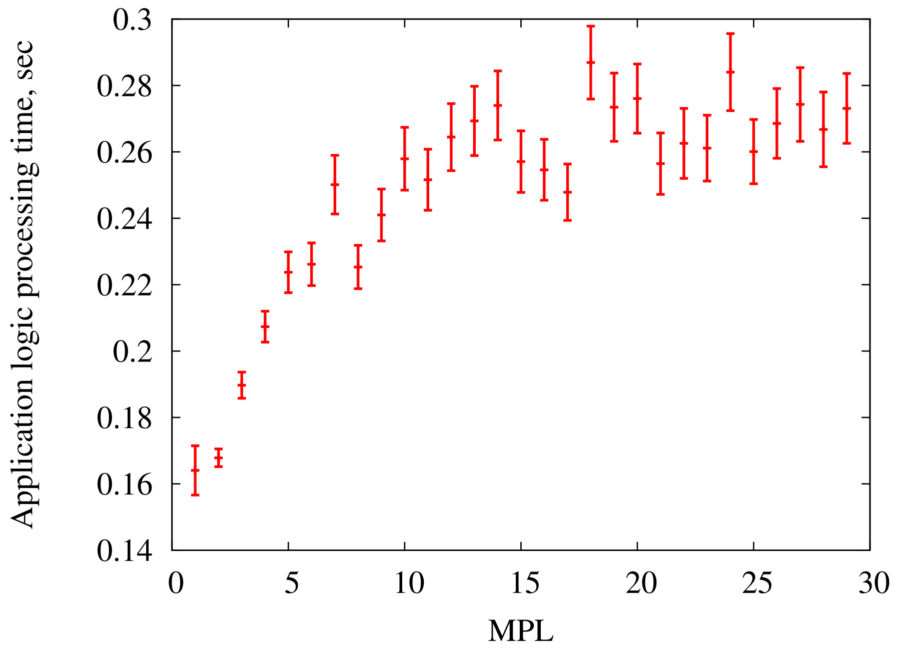

Figure 4.9: Web Service “X” application logic processing times with respect to the MPL.

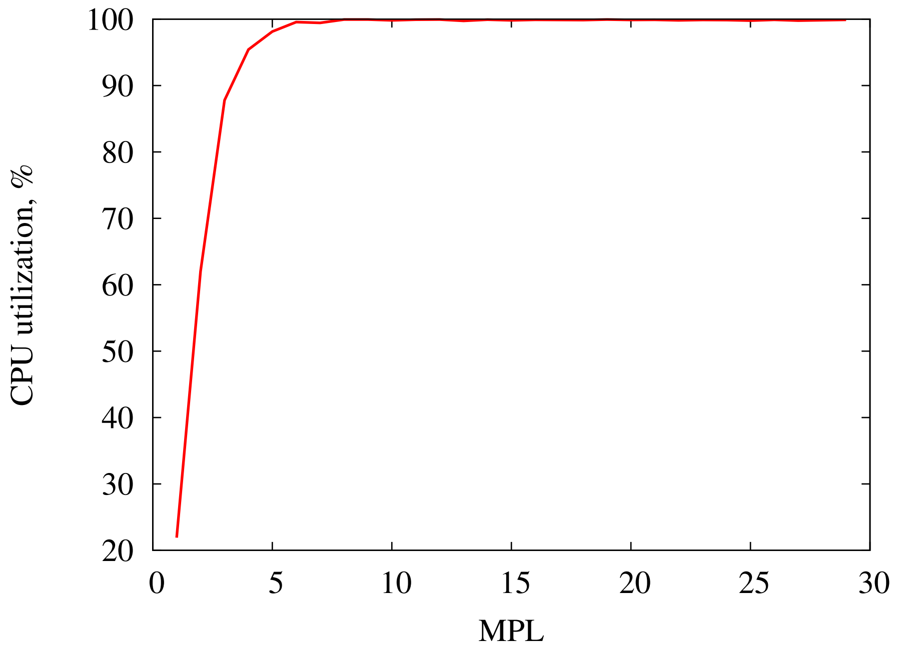

Figure 4.10: CPU utilization of the host of Web Service “X” with respect to the MPL.

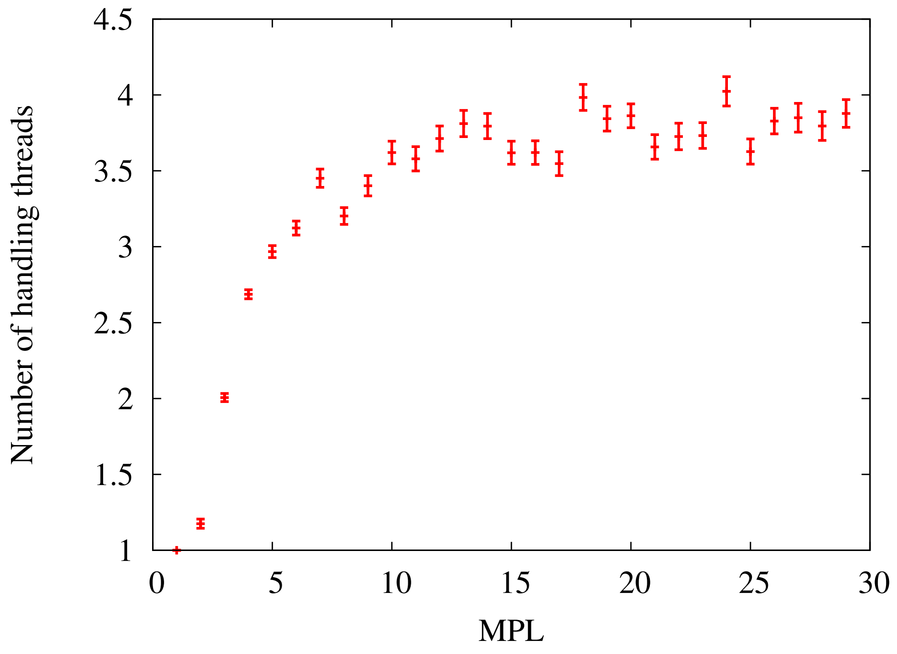

Figure 4.11: Number of used handling threads Web Service “X” with respect to the MPL. Internal WCF admission control disabled.

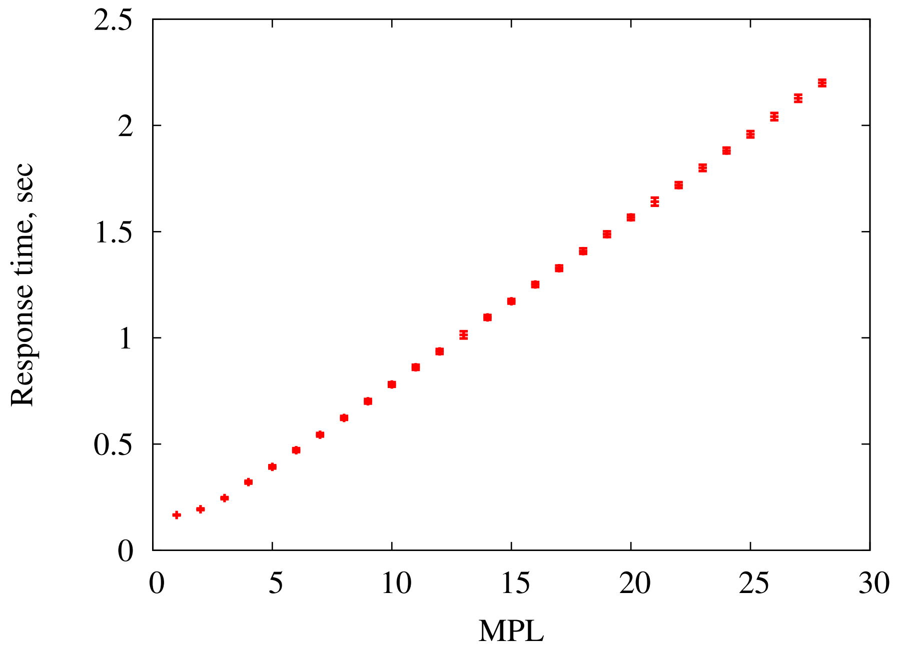

Figure 4.12: Web Service “X” response time with respect to the MPL.

Schroeder et al. [116] argue that the MPL leading to the highest throughput does not necessary result in better control over response time by means of scheduling. The higher values of the MPL contribute to increasing the number of elements accepted in service, consequently leading to a smaller number of elements remaining in the queue. Accordingly, when the MPL is increased the service time becomes larger and the waiting time declines, as it is proportional to the waiting queue size. Since non-preemptive scheduling affects only the order of elements in the waiting queue, the waiting time is the only portion of the response time which can be leveraged by a governing scheduling policy. Therefore, Schroeder et al. argue that the selection of the MPL value has to be based on the resource utilization rather than on the maximum throughput, as the slightly suboptimal choice of the MPL can result in a significantly better control at the expense of the slightly lower performance. As a result, in this scenario setting the MPL value to 5 appears as a rational trade off between the CPU utilization [Figure 4.10], throughput [Figure 4.6], application logic processing time [Figure 4.9] and response time [Figure 4.12]. Such a value of the MPL results in the response time being less than 0.5 sec, while the application logic processing time is 0.203 and the actual number of handling threads is 3.

4.3 Evaluation of Non-Preemptive Highest Value First (Static Priority Scheduling)

This section contains the evaluation of the scheduling proxy used with the Web Service described in the previous section. More precisely, this section answers the question regarding the degree of QoS differentiation achieved in practice with Static Priority scheduling.

The experiments were conducted using a load generator employing an open model, i.e. each client issues exactly one request and then leaves the system. The load was generated using the Tsung 1.3.3. load generator[4]. The clients were arriving according to a Poisson process [125], and the arrival rates were gradually increasing from 1 req/sec to 10 req/sec. The clients used the HTTP 1.0 protocol, meaning that each request resulted in opening and

[4]http://tsung.erlang-projects.org/

Table 4.1: SLAs used in the experiment with non-preemptive HVF (Static Priority scheduling).

Name	Deadline (D), sec	Price (v)	Penalty (p)
Class 1	1.0	2.0	1.0
Class 2	1.0	1.0	0.5

closing of a TCP socket. Each client was randomly chosen to pertain to either Class 1 or to Class 2 with an equal probability. SLAs for each class of clients can be found in Table 4.1. In case a request was processed before the deadline, its price is added to the profit, while each request which was completed after the assigned deadline resulted in a penalty subtracted from the profit. The assessment of compliance with the SLA was conducted on the proxy site and thus did not include the network latency, which did not exceed 1% of the service time. The waiting queue was governed by the Static Priority scheduling policy implementing HVF, which suggests prioritizing the requests with higher values. This policy was implemented by means of a priority queue, where requests are sorted first according to their values and then according to the request arrival times.

The experiment consisted of 10 phases 20 minutes each. During each phase the client arrival rate was fixed for 20 minutes. The observed metrics included response time, throughput, revenues, penalties, profit and the ratio of SLA violations. The mean values of the metrics were collected over 10 second time intervals. Revenue refers to the sum of the prices v collected from clients, while penalty is the sum of penalties. Profit is equal to the difference between the revenue and the penalties. The measured values were collected for each class of clients.

First of all, both classes have the same throughput [Figure 4.13], even though Class 1 clients have been given the priority. This can be explained by that fact that the system is not saturated and both classes have equal arrival rates. Since the system is overloaded when the request arrival rate reaches 9 req/sec, the throughput is equal to the arrival rate. However, at the point of 10 req/sec, the system starts getting overloaded, and the completion rate of the Class 2 requests begins deteriorating.

As one can notice, a few SLA violations are already present at the point where the arrival rate starts exceeding 3 req/sec [Figure 4.14]. The ratio of SLA violations does not change substantially untill the arrival rate reaches 6 req/sec. A further increase of the load leads to saturation and eventually causes a system overload. As a result, the number of SLA violations for Class 2 jumps up to 100% [Figure 4.14], which in its turn is reflected on the penalty increase [Figure 4.15], revenue decline [Figure 4.16] and consequently profit decline [Figure 4.17]. As it was expected the decline of the profit for Class 1 is significantly lower than for Class 2. The difference in those two classes emerges at the point of 6 req/sec – when the waiting queue starts building up. A further increase of the arrival rates contributes to the queue size and thus the difference becomes more substantial. An interesting observation is that when the arrival rate is 5 req/sec the total penalty is between 0 and 1 [Figure 4.15] indicating that there are only a few occasional SLA violations for both classes [Figure 4.14]. Consequently, the system is able of coping with the load of 5 req/sec without causing a substantial number of SLA violations. However, at the point of 10 req/sec, Class 1 clients generate the same 5 req/sec. Nonetheless, the percent of the SLA violations is now higher [Figure 4.14]. Such a phenomena is explained by the fact that the arriving Class 1 requests can see service busy with handling Class 2 requests.

The experiments in this section demonstrate that the scheduling proxy can be used with a real-world Web Service in order to enforce compliance with the SLAs. Such a method does not require performing any modifications on the client or service sites. Therefore, it is completely transparent to the both parties.

4.4 Web Service Model

The primary objective of this section is evaluating the effectiveness of the proposed scheduling policies. However, due to the prohibitive costs associated with setting up an environment using services and schedulers as described in Section 4.3, the evaluation will be performed using a simulation. As the global policies demand simulating service workflows, those will be modelled by means of combining a number of service models. Accordingly, in order to ensure that service workflows behaviour is the same one would expect in a real-world setting, it is of a great importance to ensure that a service model behaves exactly like a real-world service.

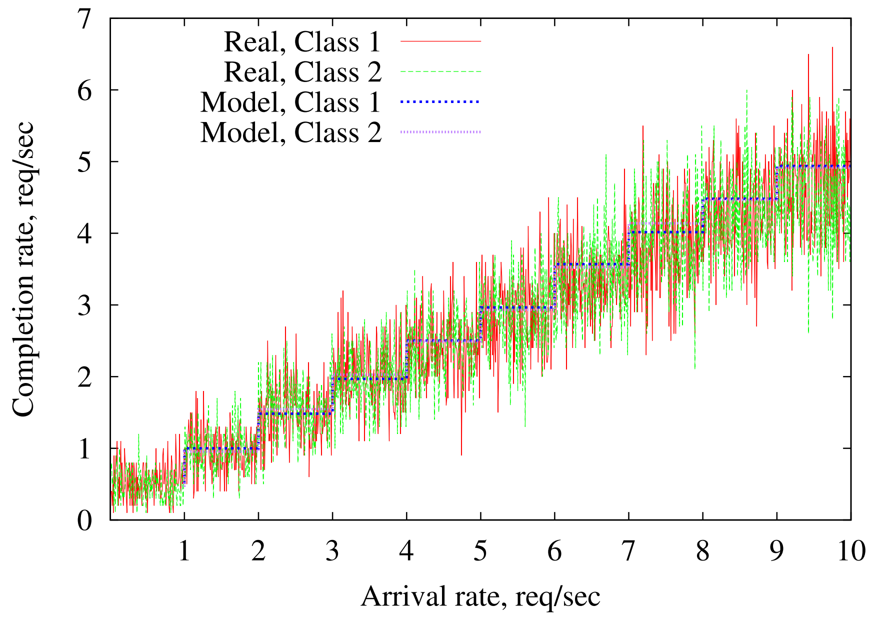

Figure 4.13: Throughput. Series marked as "Real" display the throughput measured in the experiment with Web Service "X", while series "Model" show the results of Web Service "X" simulation.

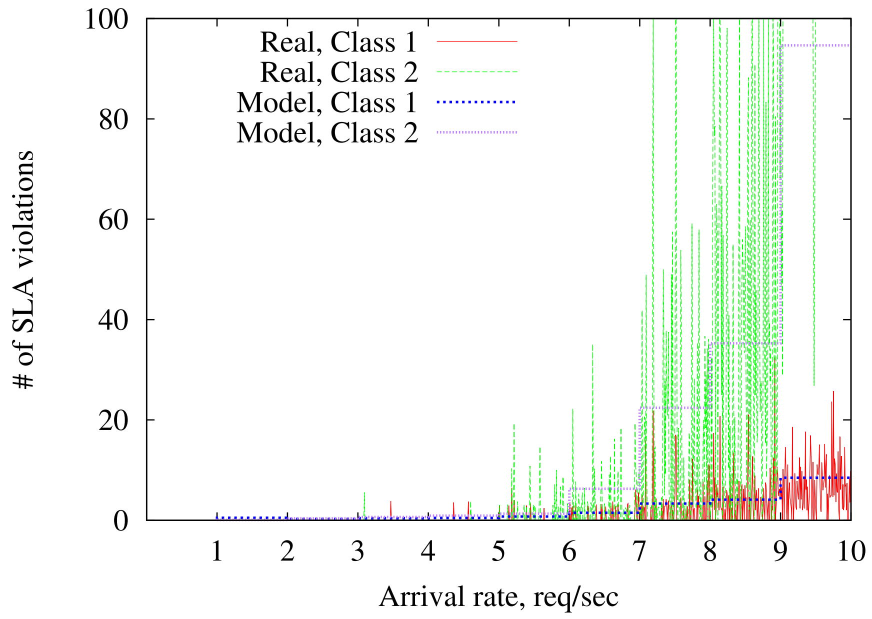

Figure 4.14: Percent of SLA violations. Series marked as "Real" display the percent of SLA violations measured in the experiment with Web Service "X", while series "Model" show the results of Web Service "X" simulation.

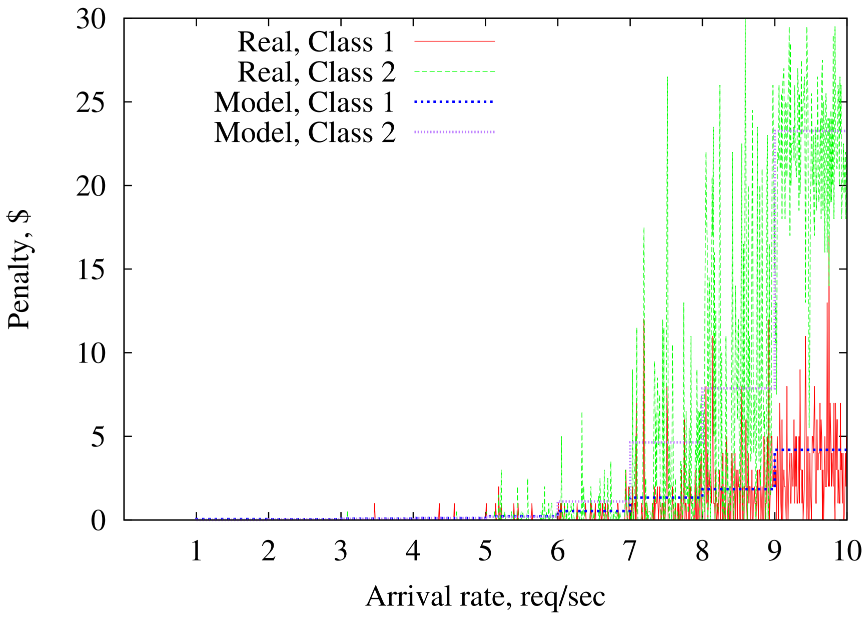

Figure 4.15: Penalties. Series marked as "Real" display the penalties measured in the experiment with Web Service "X", while series "Model" show the results of Web Service "X" simulation.

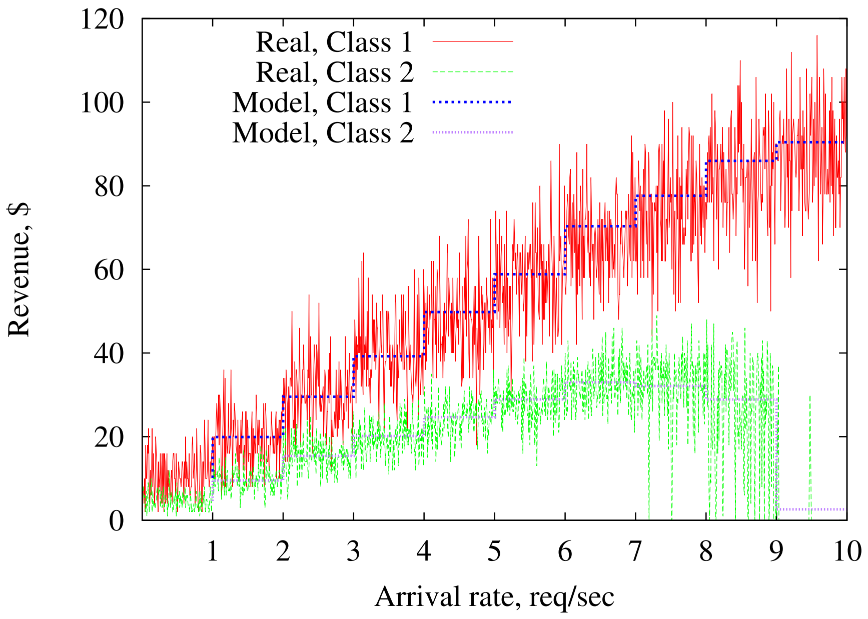

Figure 4.16: Revenues. Series marked as "Real" display the revenues measured in the experiment with Web Service "X", while series "Model" show the results of Web Service "X" simulation.

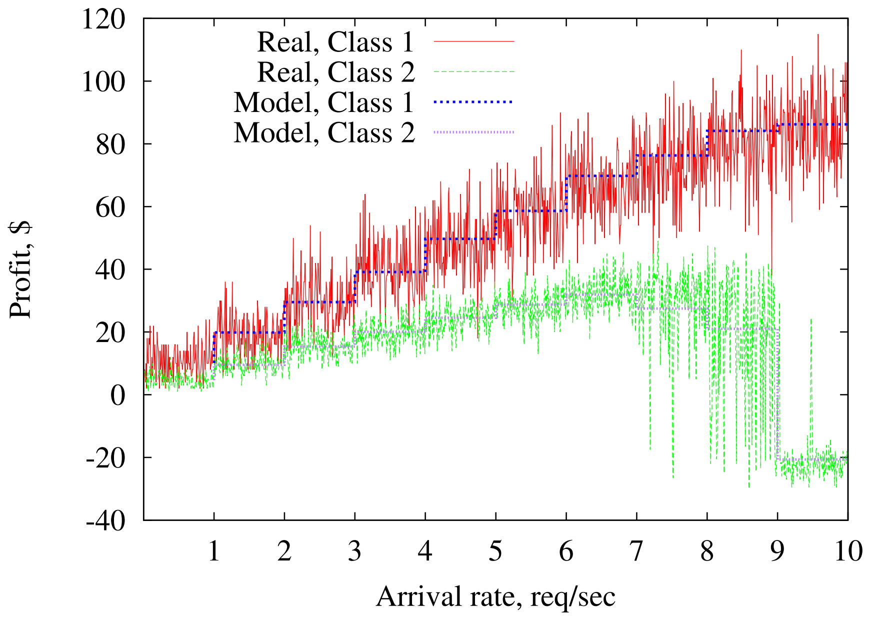

Figure 4.17: Profits. Series marked as "Real" display the profits measured in the experiment with Web Service "X", while series "Model" show the results of Web Service "X" simulation.

Due to the absence of the third-party software for simulating service workflows, an event driven simulator specifically designed for this purpose was developed. The simulator supports modeling atomic services, as well as service workflows designed using sequence, parallel split, exclusive choice and loop patterns. It was implemented in Java using Apache Commons libraries for statistical analysis[5], as well as the modified version of the engine, employed in my previous works for simulating HTTP web servers [48, 97].

The two key challenges, which emerge during any modeling process, are verification and validation. Verification refers to the process of ensuring that the model is programmed correctly, that the algorithms have been implemented properly and that the model does not contain any errors or bugs. Validation answers the question of whether the model meets its intended requirements, i.e. the entities observed within the model demonstrate the same behaviour as the ones in a real-world setting.

The verification of the simulator and its components was conducted using thorough unit tests, while the system testing was performed by employing boundary cases in which different

[5]http://commons.apache.org/

policies are expected to act identically. For example, when scheduling a single service according to the HVF policy and the service is exposed to a single class of clients using only one SLA, the service should act as the one for which the arriving requests are ordered according to FIFO.

Web Service model validation was conducted by means of contrasting the results exhibited both by a service and a model. The ordering of the requests, in model and real service cases, was performed using the HVF policy. The validation was performed against the following metrics: revenue, penalty, profit, percent of SLA violations and throughput. The simulation was conducted using an open client model. The model mimicked the setup described in Section 4.3.

One of the most common ways of modeling the behaviour of a multi-threaded server is employing a $G/G/N$ approximation [116], where N is the number of threads used for handling the incoming requests. However, an attempt to use such an approximation did not result in a sufficient proximity with the metrics measured in Section 4.3. The main reason for such a discrepancy is that the model did not reflect the fact that the actual number of simultaneously running handling threads is somewhat lower than the number of requests allowed in service by an admission control mechanism [Section 4.2]. Therefore, the decision to augment this model with an additional internal queue [Figure 4.18] was made.

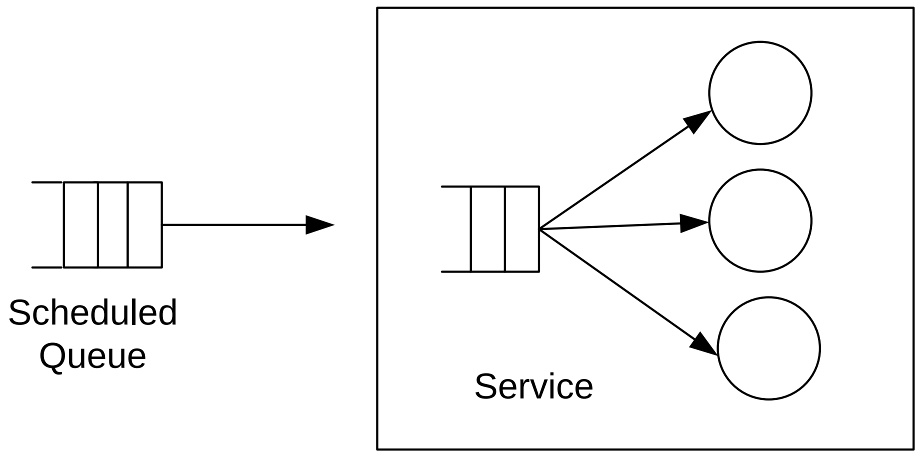

Figure 4.18: Queueing model of Web Service “X”.

After adding such an extension, various combinations of the number of servers and the maximum size of the internal queue were used to find the ones leading to the best proximity between the observed metrics. An exhaustive search among the possible values of the parameters showed that the following values result in the best fit: the number of processing servers: 3, the maximum number of requests admitted in service: 5 and the maximum length of the internal service queue: 2.

Another significant parameter in the model is the distribution of the service time. Even though the amount of CPU required for handling a request, in the modeled service, was more or less constant, using locks by the application logic resulted in a non-deterministic service time. Thus, after experimenting with various distributions and their parameters it was discovered that the best results are achieved when the service times are distributed according to a log-normal distribution with the mean of 0.202 sec and the squared coefficient of variation (SCV) of 0.5.

Charts in Figures 4.14, 4.15 , 4.16 and 4.17 show that the metrics observed in the simulation and in the experiment with the real-world service have almost identical behaviour.

4.5 Local Scheduling Policies Evaluation

The main goal of this section is investigating the efficiency of the local scheduling policies when applied to a single stand-alone service, i.e. without the global policies. The experiments in this section were conducted using the Web Service model described in Section 4.4.

The major dimensions of the experimental space consisted of the type of the request arrival process and the type of SLAs. Two types of request arrival were considered: Poisson traffic [125] and traffic where the interarrival times were log-normally distributed with the SCV of 10.0. As for SLAs, two different types were used. The first type (Correlated SLA) represents a combination of 20 SLAs, where each class of requests had a deadline inversely proportional to the price, i.e. the smaller the deadline D the higher the price v. The penalty of each request was 5 times the price, hard deadlines were 3 times longer than deadlines $\hat{D}$, and the penalty for violating a hard deadline (hard penalty $\hat{p}$) was 10 times the penalty p. The second type of SLA is where deadlines and prices do not correlate with each other. This type of SLA is regarded as an uncorrelated SLA. Note that the interdependence between deadlines and hard deadlines, prices, penalty and hard penalty remained the same. The

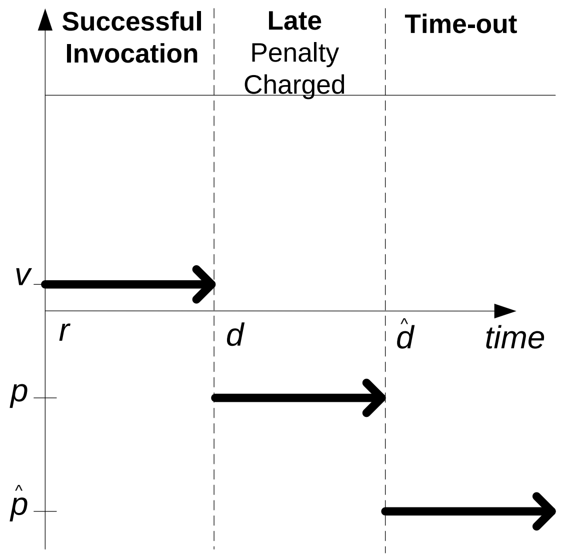

Figure 4.19: SLA model (weighted number of late jobs with time outs).

exact values of the parameters for both cases are presented in Tables 4.2 and 4.3.

The processing time of each request was assessed based on the SLAs presented in Figure 4.19. If the request is processed before time $d = r + D$, where r is the request arrival time, a service provider is rewarded with the price v. Otherwise, if the request was processed after time d it is considered to be an SLA violation and the service provider is charged p. In case the request is still waiting in the queue at instant $\hat{d} = r + \hat{D}$, it is considered to be timed out, and the request is withdrawn from the queue and abandoned and the provider is charged $\hat{p}$.

Given that each policy has specific requirements in terms of availability of information, the key question is which policy allows gaining the highest profit given the specific amount of information. The policies were evaluated both in terms of the generated profit as well as their reliability (the ratio of request which have timed out). Therefore, profit per unit of time expressed in \$/sec, was chosen as the main metric, while percent of requests which have violated the hard deadline (timed out) served as the secondary metric.

Each experiment always comprised of 20 phases. During each phase the client arrival rates stayed constant. Each phase lasted for 50,000 seconds and was repeated 12 times. The 95% confidence intervals for the observed metrics were calculated using Student's t-distribution.

Table 4.2: SLAs of type 1. Deadlines are inversely proportional to the price.

Client #	Deadline (D)	Price (v)	Penalty (p)	Hard deadline ($\hat{D}$)	Time out penalty ($\hat{p}$)
1	0.50	8.00	40.00	1.50	400.00
2	1.00	4.00	20.00	3.00	200.00
3	1.50	2.67	13.33	4.50	133.33
4	2.00	2.00	10.00	6.00	100.00
5	2.50	1.60	8.00	7.50	80.00
6	3.00	1.33	6.67	9.00	66.67
7	3.50	1.14	5.71	10.50	57.14
8	4.00	1.00	5.00	12.00	50.00
9	4.50	0.89	4.44	13.50	44.44
10	5.00	0.80	4.00	15.00	40.00
11	5.50	0.73	3.64	16.50	36.36
12	6.00	0.67	3.33	18.00	33.33
13	6.50	0.62	3.08	19.50	30.77
14	7.00	0.57	2.86	21.00	28.57
15	7.50	0.53	2.67	22.50	26.67
16	8.00	0.50	2.50	24.00	25.00
17	8.50	0.47	2.35	25.50	23.53
18	9.00	0.44	2.22	27.00	22.22
19	9.50	0.42	2.11	28.50	21.05
20	10.00	0.40	2.00	30.00	20.00

Table 4.3: SLAs of type 2. Deadlines do not correlate with the price.

Client #	Deadline (D)	Price (v)	Penalty (p)	Hard deadline ($\hat{D}$)	Time out penalty ($\hat{p}$)
1	0.50	0.78	3.88	1.50	38.75
2	1.00	3.92	19.59	3.00	195.90
3	1.50	5.07	25.36	4.50	253.61
4	2.00	2.02	10.09	6.00	100.93
5	2.50	6.88	34.39	7.50	343.86
6	3.00	0.56	2.82	9.00	28.16
7	3.50	4.68	23.38	10.50	233.76
8	4.00	4.61	23.05	12.00	230.55
9	4.50	3.39	16.95	13.50	169.46
10	5.00	7.53	37.65	15.00	376.54
11	5.50	0.87	4.35	16.50	43.47
12	6.00	7.43	37.17	18.00	371.72
13	6.50	7.40	36.98	19.50	369.76
14	7.00	4.52	22.61	21.00	226.07
15	7.50	3.55	17.77	22.50	177.66
16	8.00	3.53	17.66	24.00	176.62
17	8.50	7.51	37.53	25.50	375.34
18	9.00	3.55	17.74	27.00	177.36
19	9.50	7.73	38.63	28.50	386.29
20	10.00	7.57	37.87	30.00	378.65

4.5.1 Poisson Arrivals. Correlated SLA

As one can be observe in Figure 4.20, if the load approaches 9 req/sec the penalty starts rapidly increasing. This occurs due to the fact, that the waiting queue keeps increasing. Consequently, the waiting times begin approaching the deadlines. Another metric, revenue, shows how efficient the particular scheduling approach is with respect to dealing with the high load. Essentially, the steepness of the revenue degradation reflects the inefficiency of the particular approach, as an "ideal" scheduler would be able to maintain the steady revenue, even in the presence of the growing queue, by selecting the requests which are guaranteed to meet their deadlines. According to Figure 4.21, only EDF, LSM and HVF are performing the best according to the demonstrated penalties and revenues. Consequently, in terms of the generated profit [Figure 4.22] they also exhibit the best results. Another interesting observation is the sensitivity of the system to high loads in terms of the generated profit. Figure 4.22 shows that the profit descends much faster than it was increasing. This can be attributed to a fact that even in the "ideal" case the maximum revenue generated is fixed, as the maximum throughput of the service is a limited number, i.e. 9 req/sec, while the penalty is unbounded.

LSM is the only policy which factors in the hard deadline and the penalty associated with its violation, as a result it tries to avoid timeouts as much as it can. Accordingly, it has the lowest ratio of the timed out requests [Figure 4.24]. Surprisingly, EDF demonstrates the same result. HVF gives priority to the most expensive requests, which comes at the expense of the requests with lesser value. As a consequence, it causes occasional timeouts among cheaper requests.

As for the number of SLA violations [Figure 4.23], EDF shows the best results, due to the fact that it attempts minimising this metric, ignoring the fact that requests can have different values. The second best results are demonstrated by HVF and LSM; HRF exhibits the results worse than even FIFO and RND.

To summarise, EDF, HVF and LSM have shown impeccable results in terms of profit. In addition HVF is somewhat conservative with respect to the expensive requests, causing extra time outs of the cheaper requests.

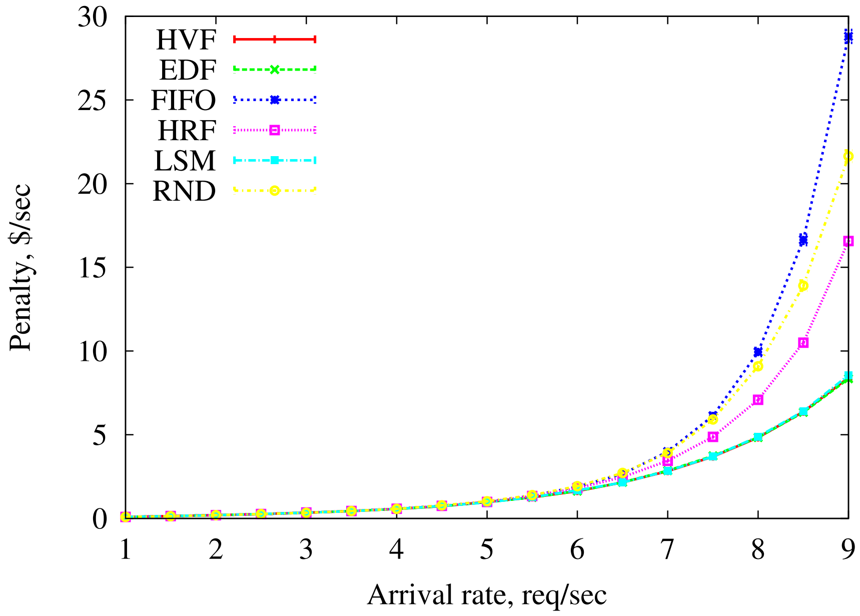

Figure 4.20: Penalty. Single service. Poisson arrivals. Correlated SLAs.

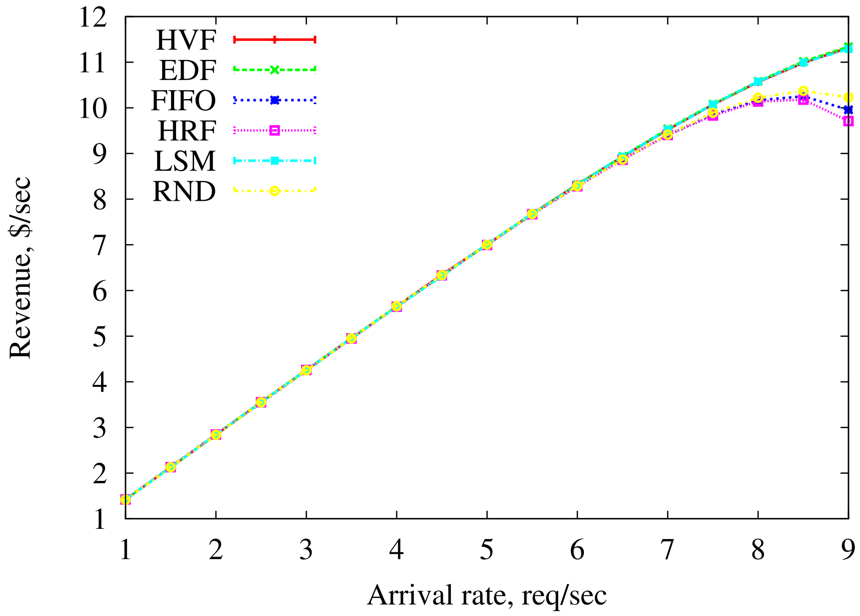

Figure 4.21: Revenue. Single service. Poisson arrivals. Correlated SLAs.

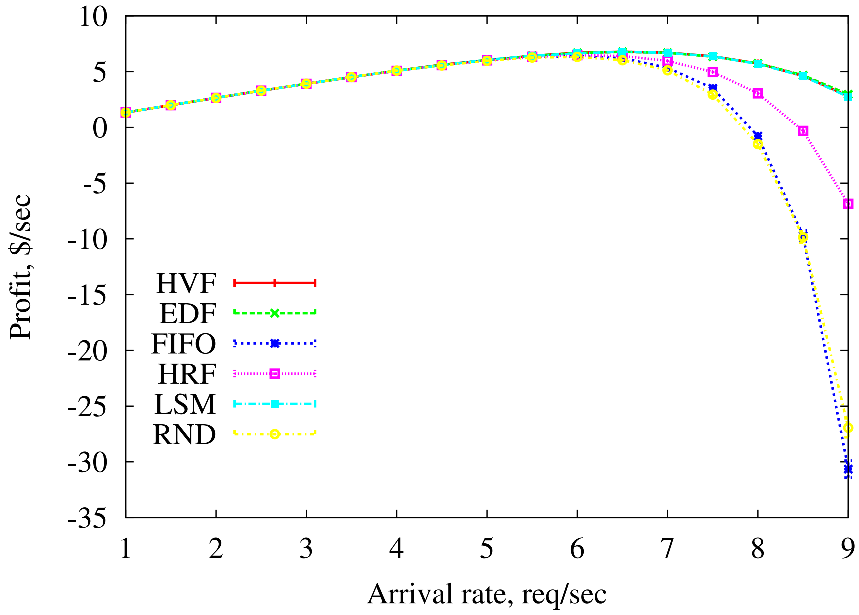

Figure 4.22: Profit. Single service. Poisson arrivals. Correlated SLAs.

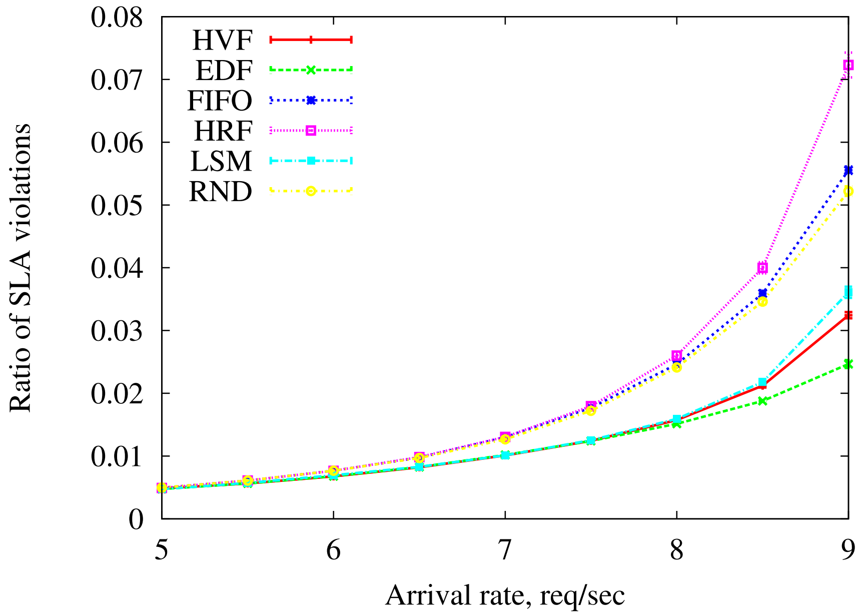

Figure 4.23: Ratio of the requests which missed their deadlines. Single service. Poisson arrivals. Correlated SLAs.

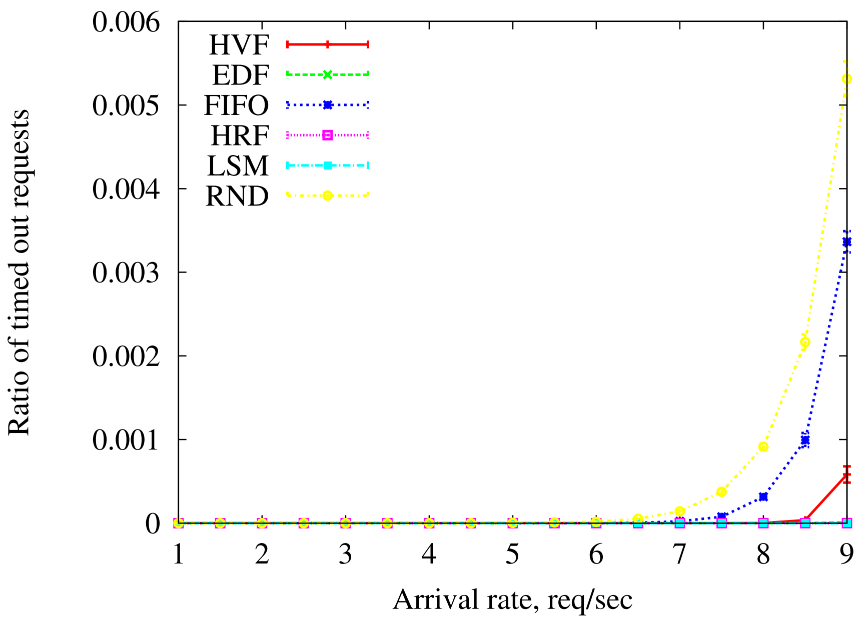

Figure 4.24: Ratio of requests which have timed out. Single service. Poisson arrivals. Correlated SLAs.

4.5.2 Poisson Arrivals. Uncorrelated SLA

In case there is no correlation between deadlines and prices, HVF is placed in a less favourable situation. The reason for this is that prioritizing the most expensive requests does not guarantee that this is the most optimal method. As a result, HVF is outperformed by LSM and EDF in terms of penalty, revenue and profit [Figures 4.25, 4.26 and 4.27]. HRF also favours expensive requests and as a result, its behaviour is quite similar to HVF.

Another interesting observation is that EDF shows slightly superior results comparing to LSM. This can be attributed to the fact that EDF tries meeting all the deadlines. In contrast to EDF, LSM is originally designed for offline scheduling, which assumes that all jobs are available at time zero. However, in this setting, there is a constant influx of newly arrived jobs which results in incorrect scheduling decisions.

As for the ratio of the SLA violations [Figure 4.29], the absolute worse policy is HVF, whose results are comparable to RND policy. This can be attributed to the fact that HVF orders requests according to their value which does not correlate with the deadline. Thus, HVF sequences requests admission in a random order. While HRF, LSM and EDF avoid time outs, which positively reflects on their penalties. LSM decisions are also impacted by

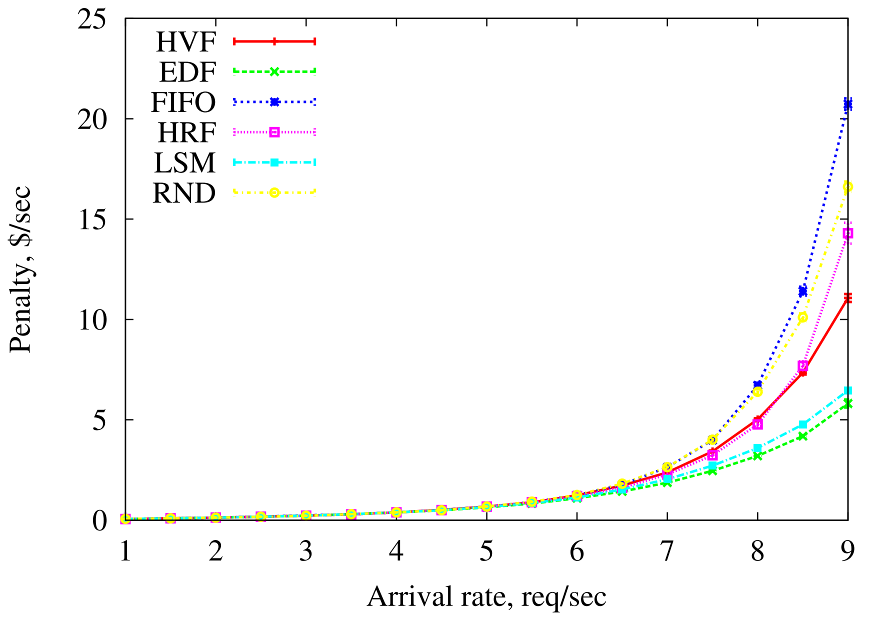

Figure 4.25: Penalty. Single service. Poisson arrivals. Uncorrelated SLAs.

the value of each request and thus it shows the ratio of SLA violations higher than EDF.

To summarize, the best results in this case are demonstrated by EDF and LSM. While HRF and HVF heuristics suffered from excessive penalties, yet they still outperform the default FIFO and RND.

4.5.3 Non-Poisson Arrivals. Correlated SLA

The major difference between this scenario and the scenario described in Subsection 4.5.1 is a difference in the request arrival process. In the scenario considered in this subsection, the arrivals are more bursty and thus the queues are longer as well. In case of the Poisson arrivals, the queue length is reaching 4.0 on average, while in this case the queue grows as big as 19 elements on average.

Figure 4.30 shows that an increase of the queue size leads to longer waiting times and higher penalties. An average increase of the penalty is 25-30%. Longer queues add additional stress on the schedulers, causing more frequent scheduling mistakes, which in their turn result in a revenue degradation [Figure 4.30].

In this scenario, HVF gives preference to more expensive requests, which also have shorter deadlines, while cheaper requests are given lower priority. As a result, the amount of cheaper

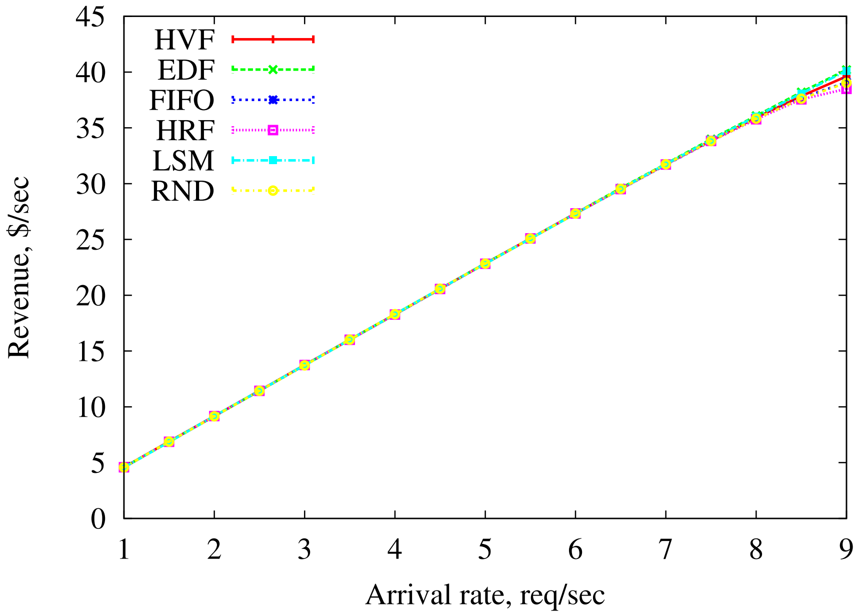

Figure 4.26: Revenue. Single service. Poisson arrivals. Uncorrelated SLAs.

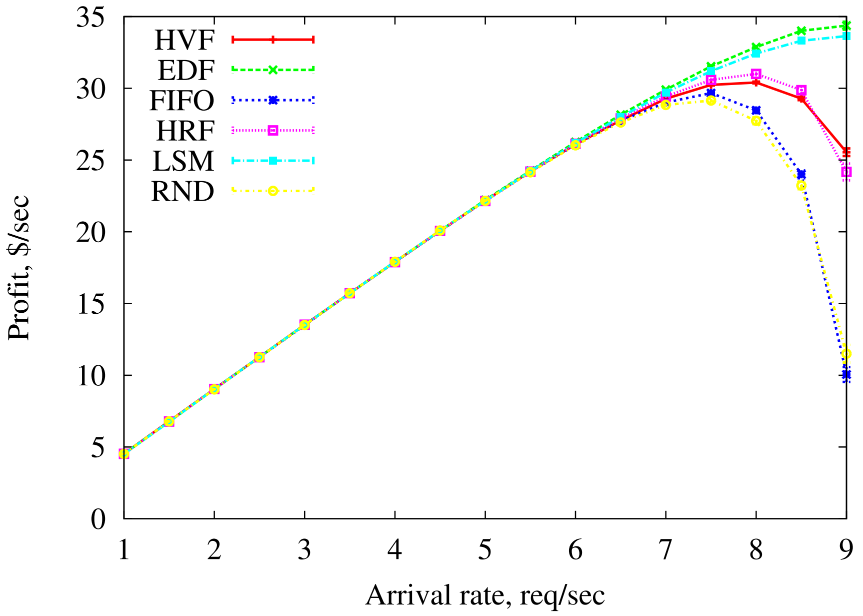

Figure 4.27: Profit. Single service. Poisson arrivals. Uncorrelated SLAs.

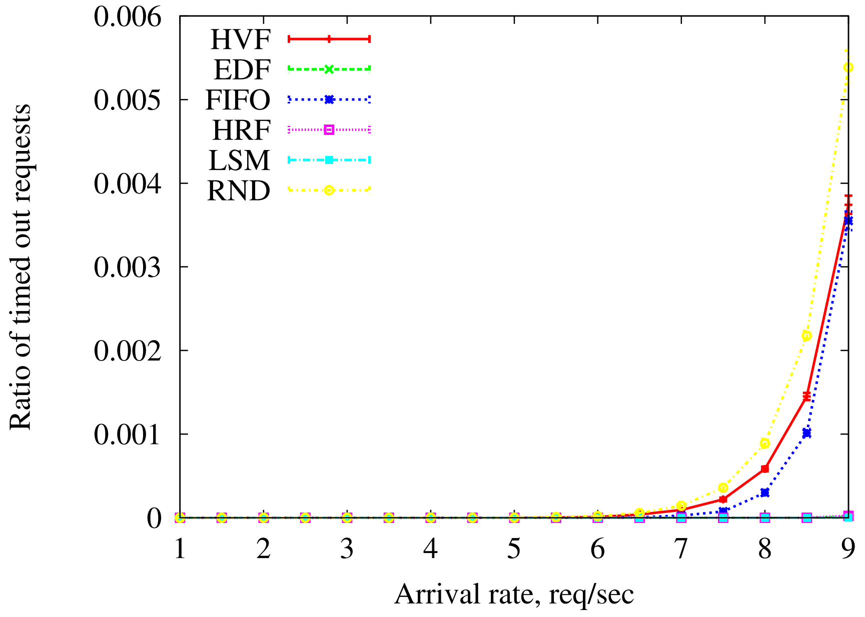

Figure 4.28: Ratio of requests which have timed out. Single service. Poisson arrivals. Uncorrelated SLAs.

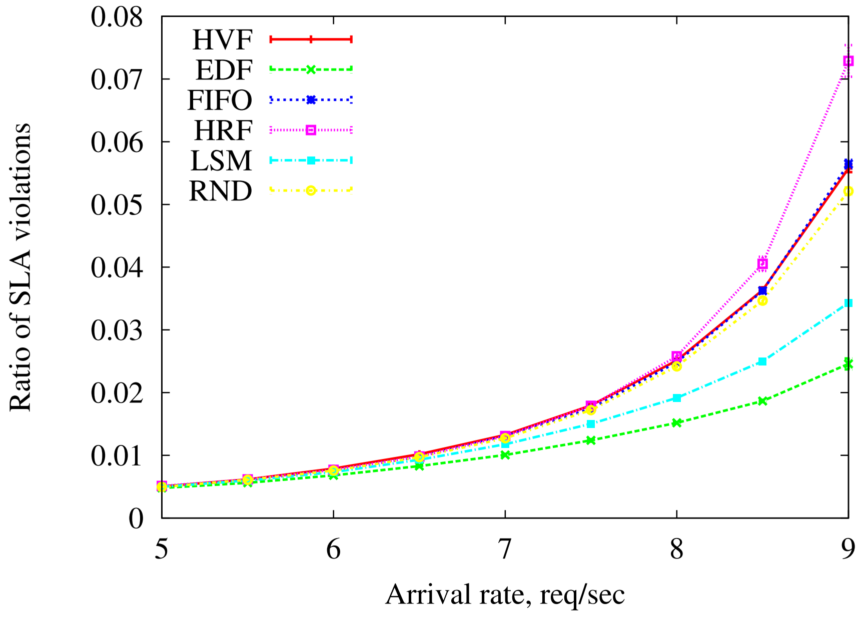

Figure 4.29: Ratio of the requests which missed their deadlines. Single service. Poisson arrivals. Uncorrelated SLAs.

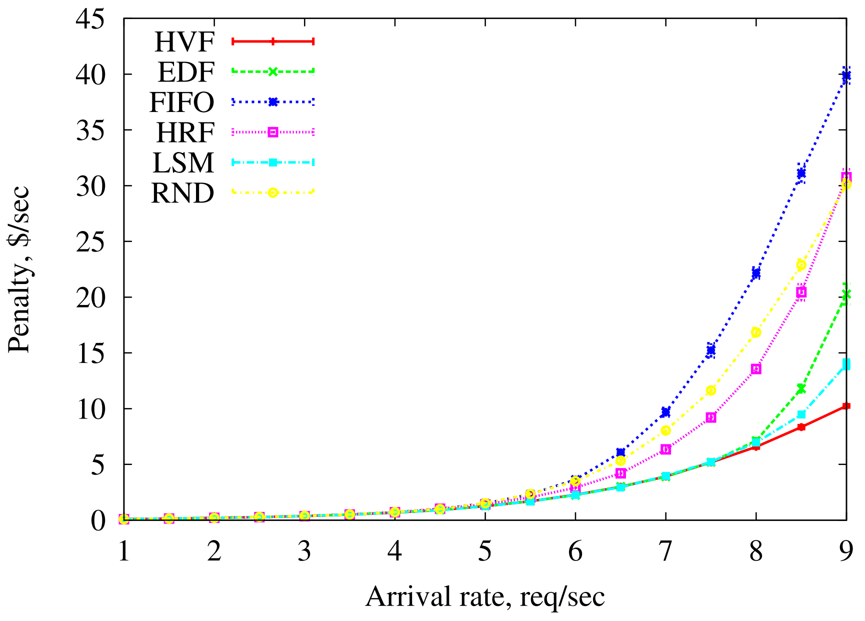

Figure 4.30: Penalty. Single service. Non-Poisson arrivals. Correlated SLAs.

requests which have timed out is quite significant when compared to other policies [Figure 4.38]. At the same time, the requests which have not timed out are more likely to meet their deadlines [Figure 4.34]. Subsequently, HVF shows a continuous increase of the revenue and a much smoother profit degradation [Figure 4.32].

LSM and HRF, have the lowest number of the timed out requests [Figure 4.38]. However, LSM's ability of meeting the deadlines becomes worse with the load increase, resulting in the number of SLA violations worse than HRF [Figure 4.34]. However, in the case of LSM, these time outs and SLA violations are more likely to occur with the cheaper requests, therefore LSM still shows the second best result in terms of penalty [Figure 4.30] and revenue [Figure 4.36]. Even though HRF and EDF do not show the results close the results demonstrated by LSM and HVF, they are still significantly outperform the default FIFO and RND policies [Figures 4.30, 4.36 and 4.32].

4.5.4 Non-Poisson Arrivals. Uncorrelated SLA

Similar to the scenario presented in the previous subsection, due to a heavy-tailed distribution of the interarrival times, there is a higher probability of the requests queueing up. As a result the mean queue size increases up to 15-17 elements.

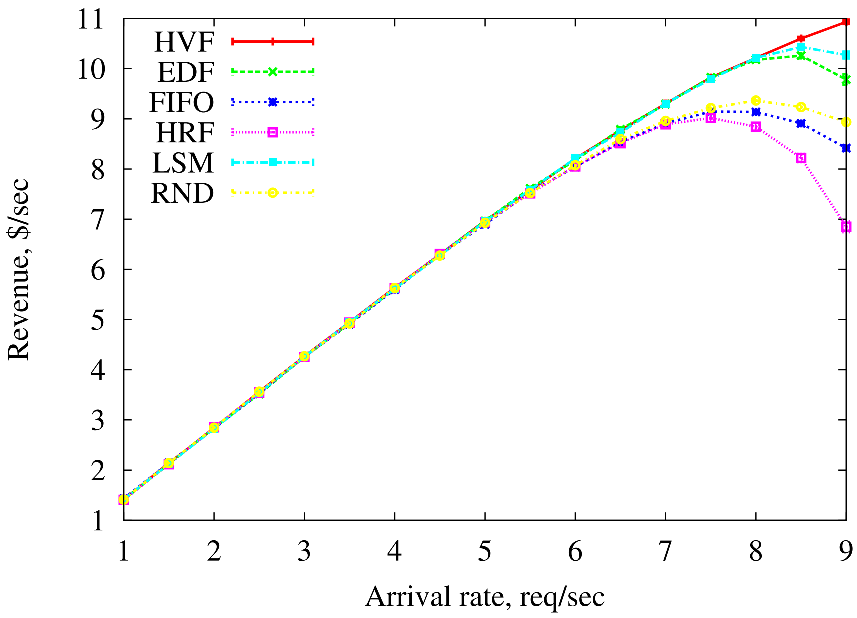

Figure 4.31: Revenue. Single service. Non-Poisson arrivals. Correlated SLAs.

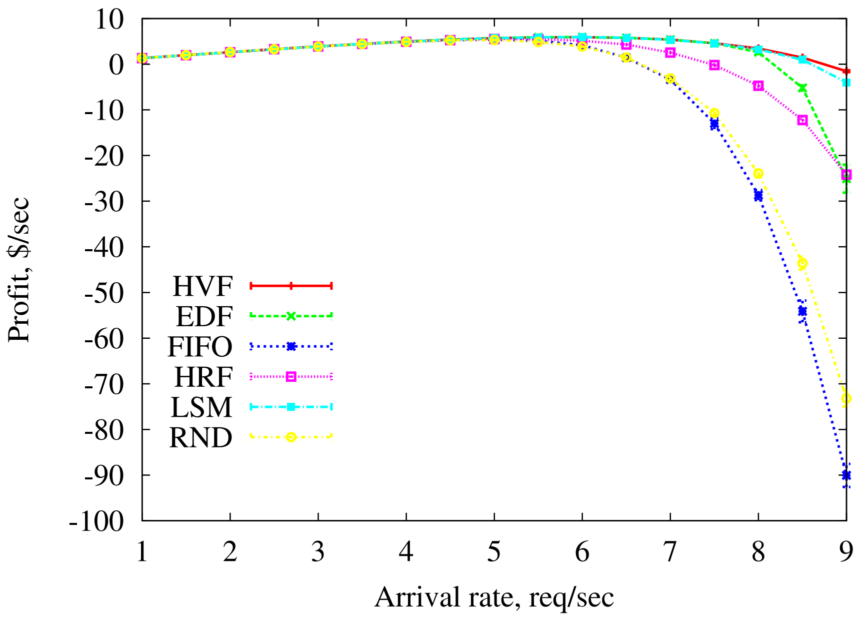

Figure 4.32: Profit. Single service. Non-Poisson arrivals. Correlated SLAs.

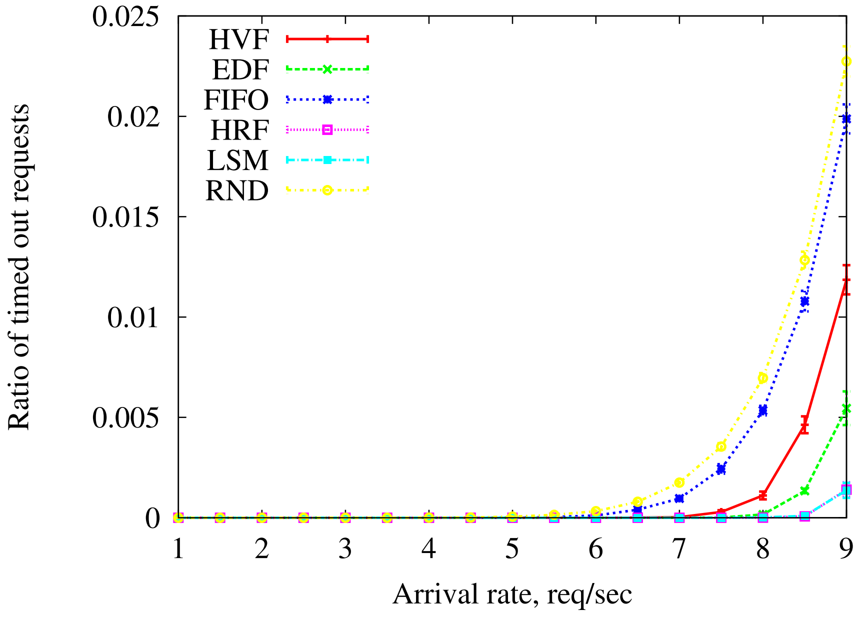

Figure 4.33: Ratio of requests which have timed out. Single service. Non-Poisson arrivals. Correlated SLAs.

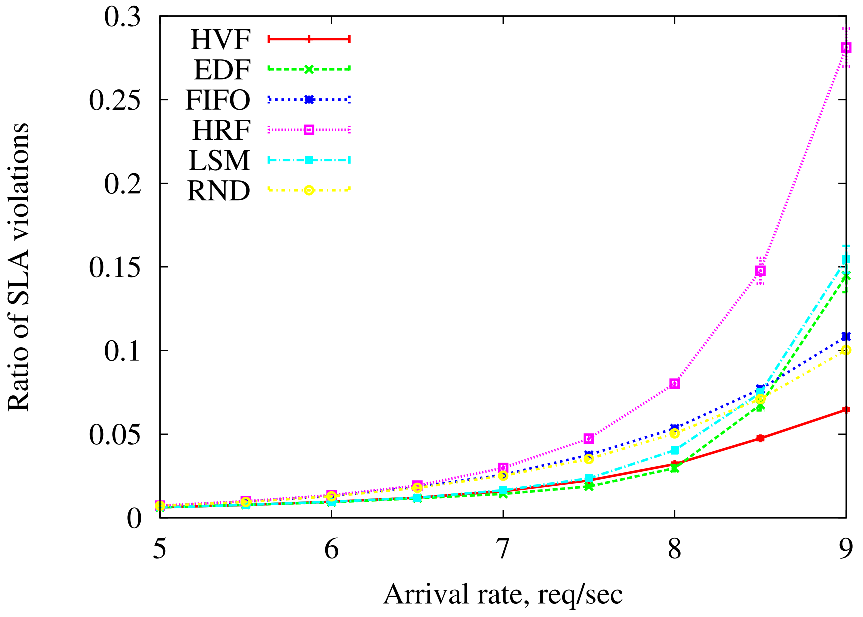

Figure 4.34: Ratio of the requests which missed their deadlines. Single service. Non-Poisson arrivals. Correlated SLAs.

The major effect of the queue increase is the longer waiting time, and accordingly the chances that requests will miss a deadline [Figure 4.38] or time out [Figure 4.39] are significantly higher than in the case of Poisson arrivals [Section 4.5.2]. The latter is a determining factor for the penalty [Figure 4.35], revenue [Figure 4.36] and consequently the profit [Figure 4.37].

As in the previous case, the schedulers are now facing a more challenging problem due to the longer waiting queue. Therefore, the deterioration of the revenue, reflecting the efficiency, is more pronounced after the load exceeds 8 req/sec. As it can be seen in Figure 4.36, the only policy which shows a steady revenue increase is HVF, while the revenue of other policies starts degrading, eventually reaching lower values than FIFO. What is even more interesting, is that EDF and LSM, as in the case of Poisson arrivals show the best results, till the moment when the traffic starts exceeding 8-8.5 req/sec [Figure 4.37]. At this point, the waiting queue contains 4.5 requests on average, which corresponds to a Poisson case with an arrival rate of 9 req/sec. As soon as the queue starts accumulating more requests the performance of these policies decreases. This is indicative of EDF being susceptible to the "domino" effect. As for LSM, most of the losses occur with the requests having shorter deadlines, as most of the penalties originate due to the SLA violations [Figure 4.39] rather than due to time outs [Figure 4.38]. This phenomena can be attributed to the fact that LSM tends to postpone requests until later, as this appears as an optimal solution at the moment. Therefore, the requests are usually served very close to their deadlines, while arrivals of new requests often compel changing the schedule, while any changes would result in additional deadline violations. Accordingly, the policy is not cautious enough to make decisions in the presence of unexpected request arrivals, thus the most significant losses occur with clients having shorter deadlines, which are the most sensitive to changes in the schedule.

4.5.5 Summary

The proposed local scheduling policies, namely HVF, EDF, HRF and LSM, were evaluated under different traffic conditions (Poisson and non-Poisson bursty traffic) in combination with two different types of SLA. The SLAs of the first type assumed a strong correlation between the values of the requests and the deadlines associated with them [Table 4.3]. Meanwhile,

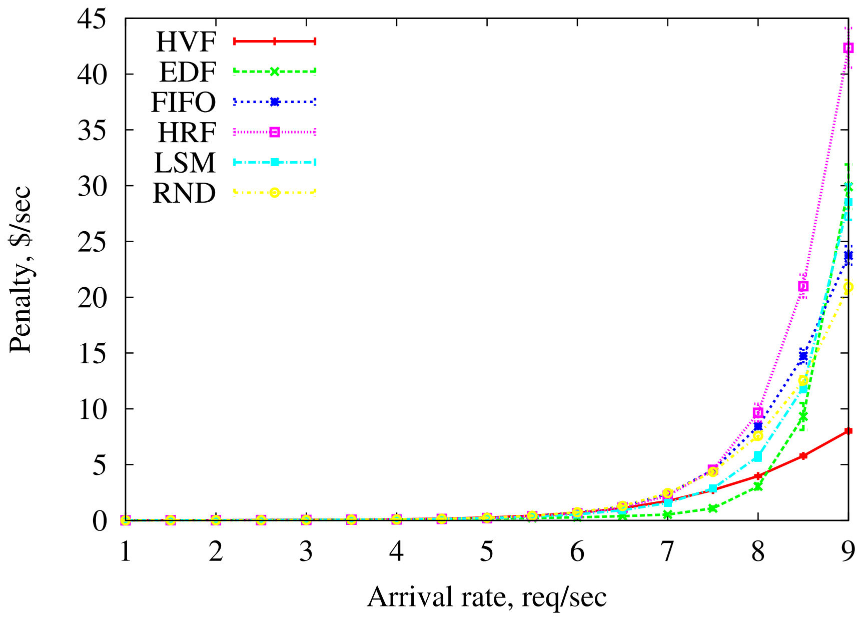

Figure 4.35: Penalty. Single service. Non-Poisson arrivals. Uncorrelated SLAs.

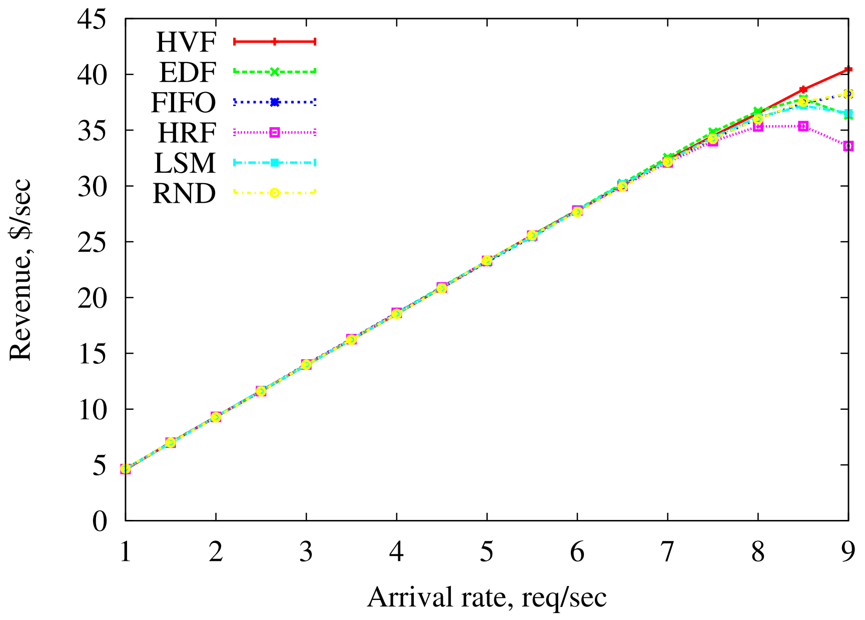

Figure 4.36: Revenue. Single service. Non-Poisson arrivals. Uncorrelated SLAs.

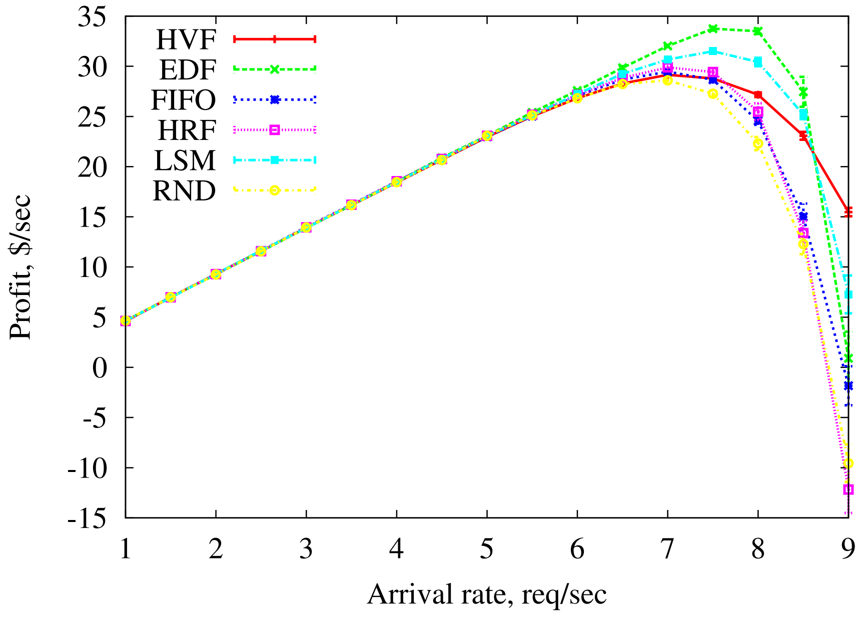

Figure 4.37: Profit. Single service. Non-Poisson arrivals. Uncorrelated SLAs.

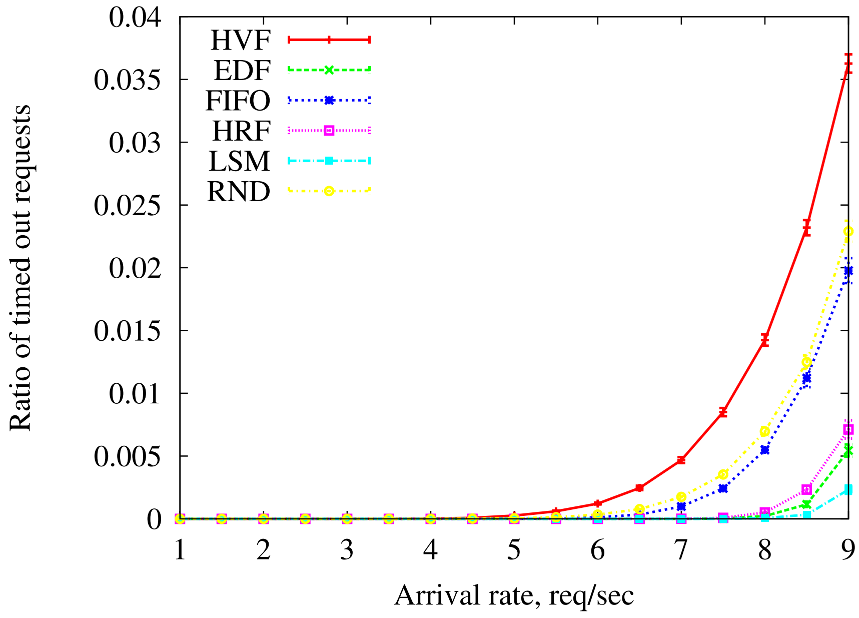

Figure 4.38: Ratio of requests which have timed out. Single service. Non-Poisson arrivals. Uncorrelated SLAs.

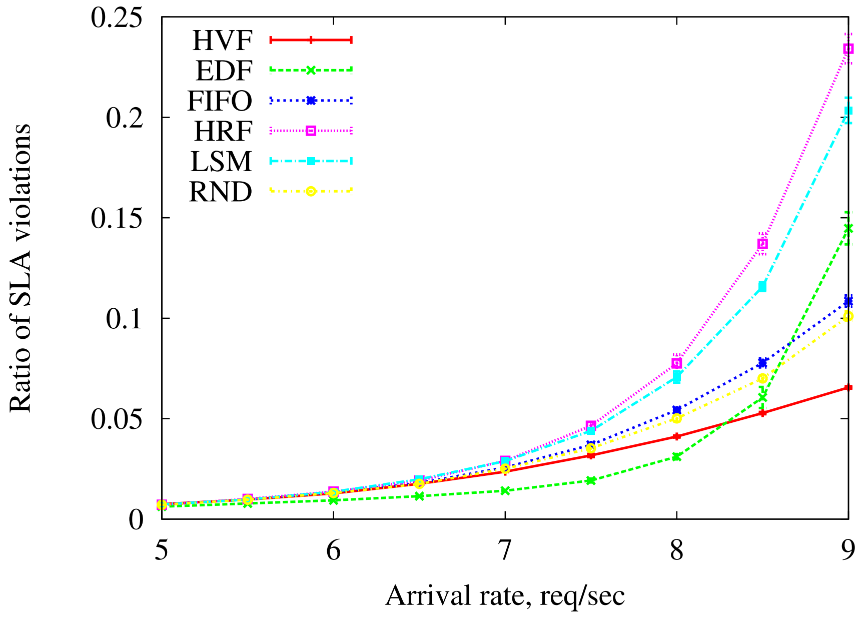

Figure 4.39: Ratio of the SLA violations. Single service. Non-Poisson arrivals. Uncorrelated SLAs.

the SLAs of the second type contained the agreements where deadlines and values do not depend on each other [Table 4.3]. Besides the proposed local scheduling policies, RND and FIFO policies were also considered for comparative purposes.

LSM exhibits the most consistent results which are usually at least second best. Yet, in the case of a very high load, it is prone to performance degradation. Note that in all cases it is superior to RND or FIFO. HVF shows the best results when used with correlated SLAs, but the results are not as strong when there is no correlation between the deadline and the value in the SLAs. Even though, the gains from HRF are not the best, they are consistent in three out of four cases. The policy shows the profit comparable to RND in the scenario with bursty traffic and uncorrelated SLAs. EDF, without directly addressing the scenarios where requests have different values, shows the results close to the best in case of the correlated SLAs.

4.6 Global Scheduling Policies Evaluation

The main goal of the simulation, presented in this section, is benchmarking the proposed policies in a setting mimicking a service infrastructure used by a more or less typical medium business. It is assumed that the organization is providing composite services exposed via predefined SLAs. Profit generated by the organization is used as the main comparison criterion. Reliability was also considered, as the secondary metric.

The considered setting comprised 100 services, were each service was hosted on a separate machine. Each service had the same properties of jobs and the value of the MPL, as the service described in the previous section. Atomic services were combined in 40 workflows which were randomly generated. Generation of the workflows was performed by means of a recursive procedure and the resulting workflows had the following properties:

- Sequence segment. The maximum number of segments in each sequence segment was no greater than 4, meanwhile the actual number was generated using a uniform distribution.
- Loop segment. Each loop performed at most 4 invocations of its corresponding subsegment, the number of iterations was uniformly distributed.
- Exclusive choice segment. In case of an exclusive choice, the number of branches was distributed according to a uniform distribution on the interval $[1 \ldots 4]$, while the probabilities of invoking specific branches were equal.
- Parallel split segment. Each parallel split segment had at least 1 and at most 4 branches, and the number of the branches was generated using a uniform distribution. The maximum depth of the tree consisting of segments was set to six.

The hierarchical structure was generated recursively, where the type of each segment was randomly selected. The exact algorithm used for generating the workflows is presented below.

```
procedure GENERATE_RANDOM_SEGMENT(currentdepth, maxdepth)
if currentdepth = maxdepth then
    return atomic segment invoking an atomic service chosen randomly
```

```
else
  dice:= random(0,99) {where random is a function for generating uniformly distributed
  random numbers, the parameters indicate the lower and the upper bounds}
  if dice in [0..39] then
    return an atomic segment invoking an atomic service chosen randomly
  end if
  if dice in [40..54] then
    k:= random(1,4)
    return a sequence segment with a number of segments equal k, each subsegment is
    generated by GENERATE_RANDOM_SEGMENT(currentdepth+1, maxdepth)
  end if
  if dice in [55..69] then
    k:= random(1,4)
    return a loop segment with a number of the number of repetitions equal to k, each
    subsegment is generated by GENERATE_RANDOM_SEGMENT(currentdepth+1, maxdepth)
  end if
  if dice in [70..84] then
    k:= random(1,4)
    return an parallel split segment with a number of branches equal to k, each subseg-
    ment generated by GENERATE_RANDOM_SEGMENT(currentdepth+1, maxdepth)
  end if
  if dice in [85..99] then
    k:= random(1,4)
    return an exclusive split segment with a number of branches equal to k, each subseg-
    ment generated by GENERATE_RANDOM_SEGMENT(currentdepth+1, maxdepth),
    the probability of selecting each branch is 1/k
  end if
end if
```

Each composite service was consumed by 10 clients, where each client had a customized SLA. Initially clients were issuing requests at the rate set to 0.05 req/sec. Each experiment consisted of 20 phases. Each phase corresponded to 50,000 seconds of the simulated time and was repeated 12 times. The arrival rate was increased by 0.0167 req/sec (33.33% of the initial arrival rate) within each phase. The 95% confidence intervals of the observed metrics were calculated using Student's t-distribution.

Twelve scheduling policies were considered: EDF+EF, EDF+PF, EDF+UD, LSM+EF, LSM+PF, LSM+UD, HRF+EF, HRF+PF, HRF+UD, HRF+EF and HVF. While FIFO and RND were also added for comparison purposes, where FIFO refers to the First-In-First-Out scheduling policy and RND is a scheduling policy organizing the waiting queues in a random order. Note that in the discussion of the results LSM+* denotes LSM+UD, LSM+PF and LSM+EF policies, accordingly EDF+* will correspond to EDF+EF, EDF+PF and EDF+UD polices. The same notation rule will be also applied to the HRF policy.

The main dimensions of the experimental space were the type of traffic and the type of SLAs. As the for the traffic, it was either Poisson or heavy-tailed with the SCV of 10.0. Meanwhile, the first type of SLAs – correlated SLAs – contained agreements where shorter deadlines corresponded to higher prices. In case of the uncorrelated SLAs, prices and deadlines were completely independent.

The x-axis on the charts show the request arrival rate per client. Note that in order to improve the readability, the charts were truncated and do not contain the parts where the system was overloaded and where the policies showed exactly the same results.

4.6.1 Poisson Arrivals. Correlated SLA

In this scenario the traffic was assumed to be Poisson, i.e. composite service requests' arrivals are independent and the requests' interarrival times follow an exponential distribution. Also it was assumed that the deadline, hard deadline, penalty and hard penalty were determined by the price of a request. More precisely, in this case for each client i ($i = 1 \dots 10$) of each workflow j ($j = 1 \dots 40$) the deadline was set to $D_{ij} = i * 2$, while the other parameters of the SLA were expressed through the deadline in the following way: $v_{ij} = 4.0/D_{ij}, p = 5 * v_{ij}, \hat{p_{ij}} = 10 * p_{ij}, \hat{D_{ij}} = 3 * D_{ij}$.

The profit generated by the EDF+* policies can be seen on a chart in Figure 4.40. Even though EDF+PF shows somewhat worse results than EDF+EF and EDF+UD at the point

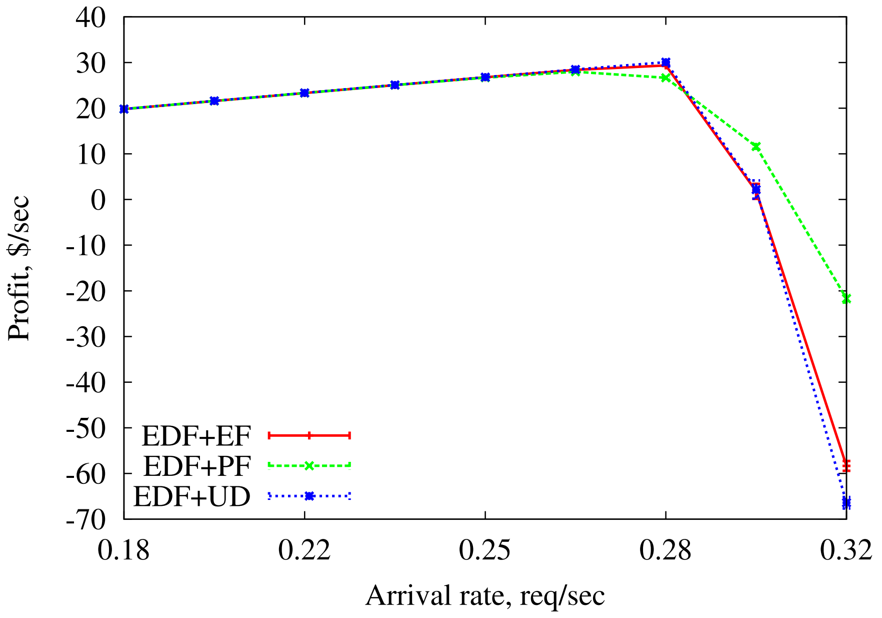

Figure 4.40: Profit. Medium business. Poisson arrivals. Correlated SLAs. EDF+*.

of 0.28 req/sec, the benefits of using PF become more evident when the load approaches 0.32 req/sec. Surprisingly, the difference in the performance of EF and UD appears to be quite insignificant.

In case of LSM+* policies [Figure 4.42], PF shows the absolute best results. This can be explained by the fact that PF more properly allocates the deadlines to component service invocations. A quite unexpected result is that EF is being outperformed by UD, also the difference in profit between those two policies is more substantial than in the previous case.

As for the HRF case [Figure 4.42], PF also appears to show the strongest results, while EF and UD behave almost identically.

The chart showing all the profits by all policies is presented in Figure 4.43. As it can be seen, LSM+PF is superior to the other policies. Note that all of the proposed policies show the results stronger than FIFO and RND. Since in this case higher values correspond to shorter deadlines, using HVF yields to strong results. In terms of reliability [Figure 4.44] all policies behave without any significant differences. To conclude, the best five policies include LSM+PF, HVF, EDF+PF, HRF+PF and LSM+EF, meanwhile PF appears to be the best global scheduling rule.

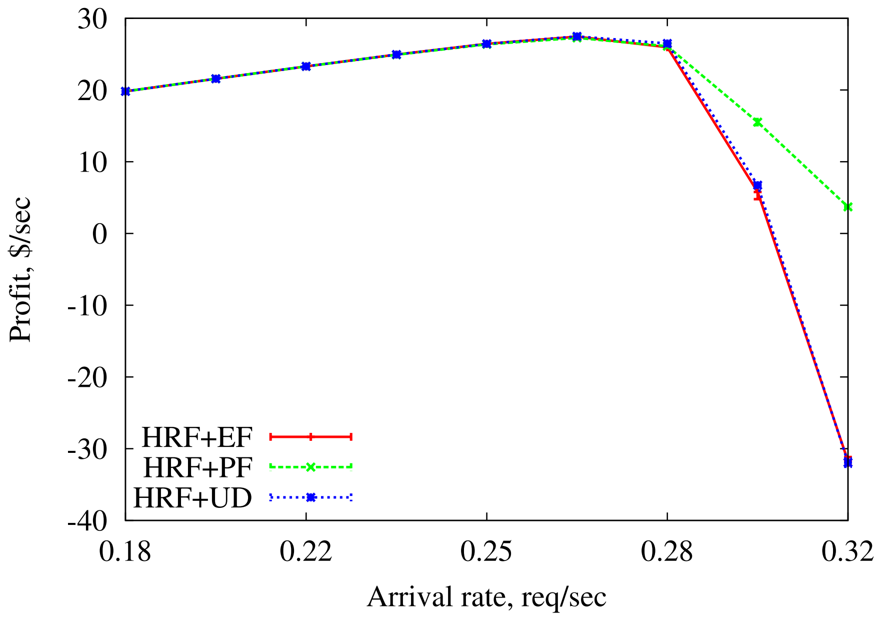

Figure 4.41: Profit. Medium business. Poisson arrivals. Correlated SLAs. HRF+*.

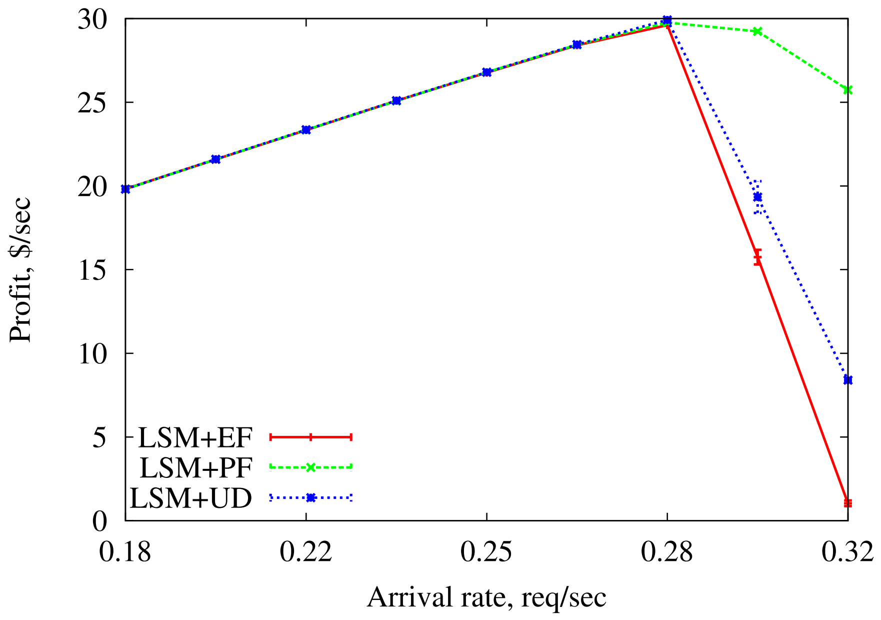

Figure 4.42: Profit. Medium business. Poisson arrivals. Correlated SLAs. LSM+*.

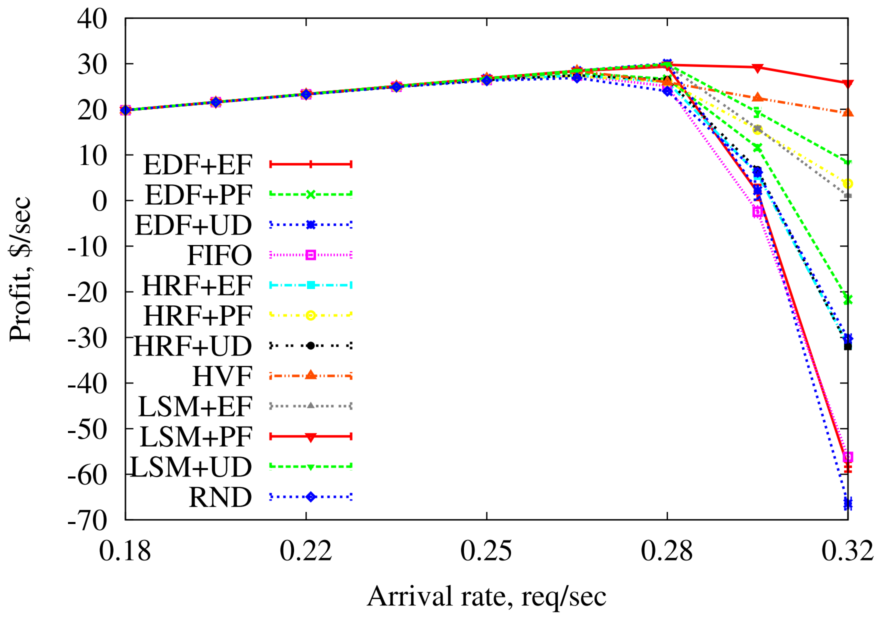

Figure 4.43: Profit. Medium business. Poisson arrivals. Correlated SLAs.

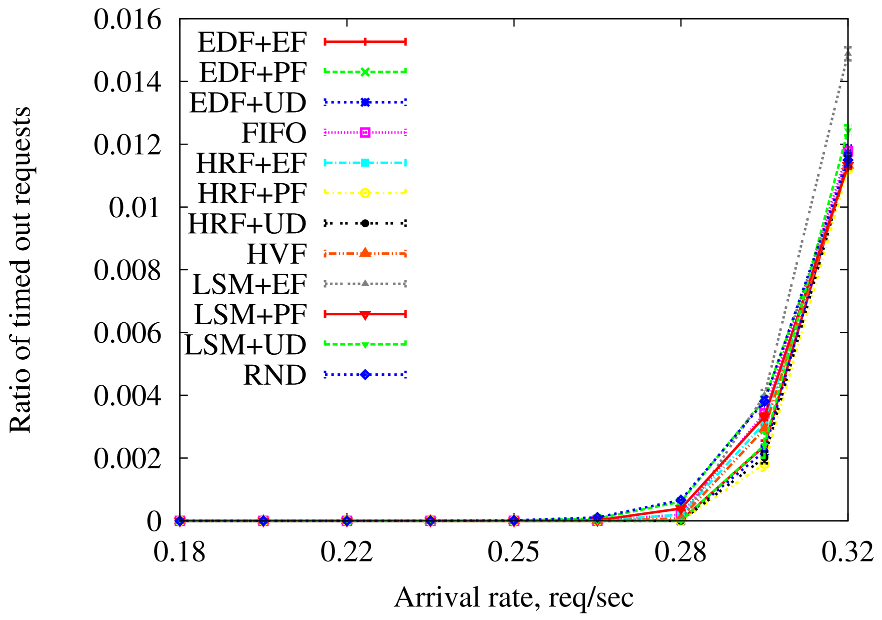

Figure 4.44: Ratio of requests which have timed out. Medium business. Poisson arrivals. Correlated SLAs.

4.6.2 Poisson Arrivals. Uncorrelated SLA

In this case the traffic was Poisson and the SLAs were uncorrelated, i.e. hard deadlines were determined by deadlines, while price, penalty and hard penalties were expressed through each other, yet did not depend on any of the deadlines. More precisely, each workflow j needed to complete a request from the client i in $D_{ij} = i * 2$ seconds. Meanwhile, the price v for completing requests on time was uniformly distributed on the interval [0.4. . . 8]. The remaining parameters of the contract were determined by the deadline and the price: $p_{ij} = 5 * v_{ij}, \hat{p_{ij}} = 10 * p_{ij}, \hat{D_{ij}} = 3 * D_{ij}$.

Figure 4.45 shows that the behaviour of the EDF+* policies is not affected by the correlation between the deadlines and the values of the requests. Besides that, the EF and UD versions behave almost identically (same as in the previous case). As in the previous scenario, the drop in the efficiency of the PF scheme is less marked than the one by UD or EF. A smaller drop can be explained by the fact the EDF does not differentiate the requests according to their values, and since there is no correlation between the deadlines and the prices EDF is put in a more favourable situation.

In case of LSM [Figure 4.46], the trend is almost identical to the one presented in the previous subsection. At the point of 0.28 req/sec PF shows slightly worse results (3-5% difference) than UD and EF, while with the increase of the load PF exhibits a much slower profit deterioration than the other global scheduling policies. With the load reaching 0.32 req/sec, the profit generated by the LSM governed system is 175. At the same time, in case EF and UD are used, the profit drops to near zero values. As in the previous case, EF and UD behave almost identically in terms of the main metric.

Figure 4.47 indicates that the absence of the relations between the deadlines and the prices (penalties) affects HRF+* policies in a more drastic way. All three modifications show a marked profit degradation with the increase of the load. PF shows the best results, except for the case when the load reaches 0.32 req/sec, as in that case the profit drops below EF by 25%.

Figure 4.48 contains the results produced by all the policies. In contrast to the previous case, only three policies - LSM+PF, EDF+PF and HVF demonstrate the profit higher than

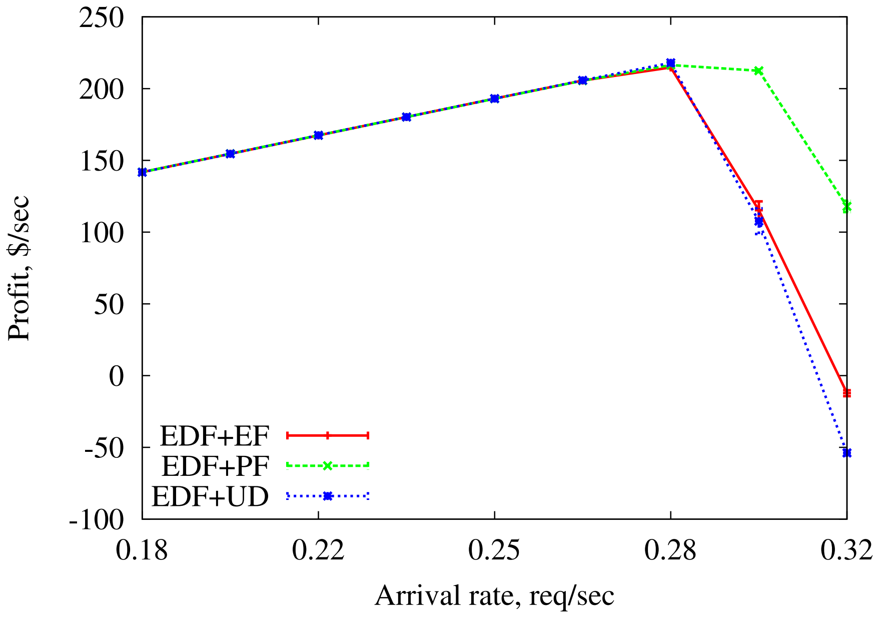

Figure 4.45: Profit. Medium business. Poisson arrivals. Uncorrelated SLAs. EDF+*.

FIFO, while the best policy, in this scenario, is LSM+PF. As for the reliability [Figure 4.49] LSM+PF and HVF have somewhat higher ratio of the timed out requests, when the load is between 0.25 and 0.3 req/sec, while if the load increases to 0.32 req/sec they exhibit the same results as the other policies.

In terms of the reliability [Figure 4.49], all policies behave almost identically, apart from LSM+PF and HVF. What is even more interesting is that those two policies, are the best policies in terms of the profit.

4.6.3 Non-Poisson Arrivals. Correlated SLA

In this scenario the traffic is assumed to be more heavy-tailed than in the previous two cases, the interarrival times were distributed according to a log-normal distribution with a squared coefficient of variation of 10.0.

The profits generated by EDF+* policies are reported in Figure 4.50. Similar to the case with a single service, traffic variability has a direct impact on the lengths of the waiting queues. The relations between the policies performances is preserved, meaning that PF shows the best results during the high load (0.32 req/sec) and slightly loses to EF and UD when the load in the range of 0.25-0.3 req/sec. However, even though PF appears to be much stronger

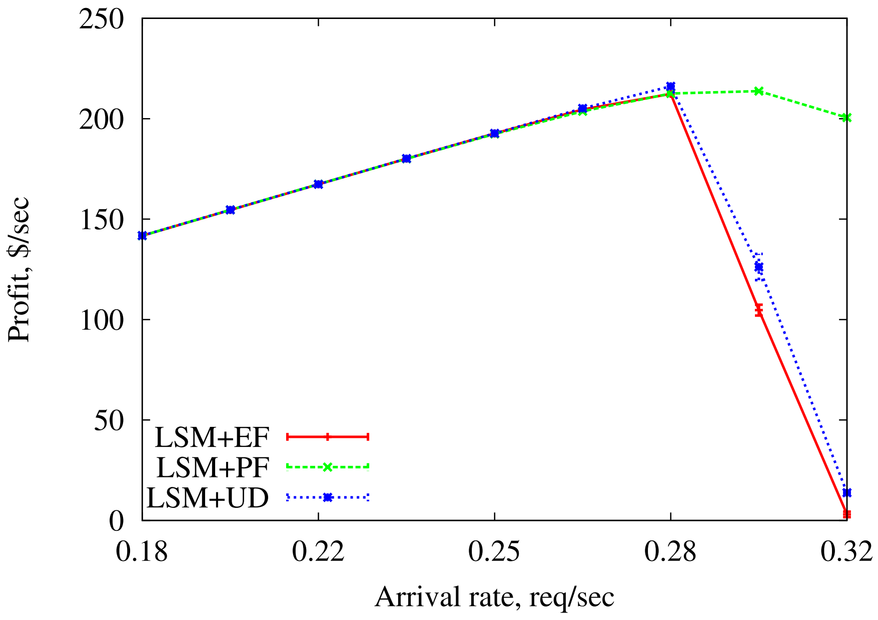

Figure 4.46: Profit. Medium business. Poisson arrivals. Uncorrelated SLAs. LSM+*.

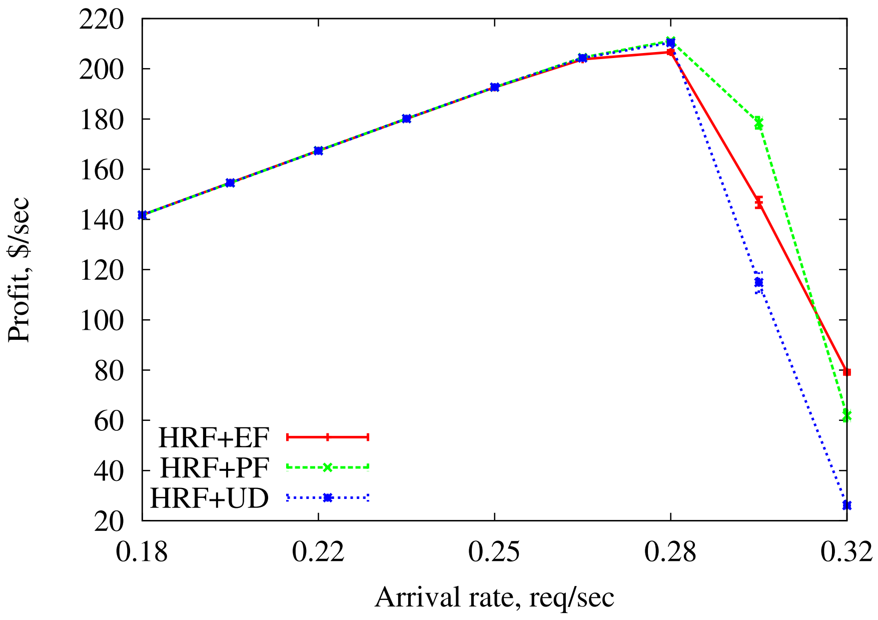

Figure 4.47: Profit. Medium business. Poisson arrivals. Uncorrelated SLAs. HRF+*.

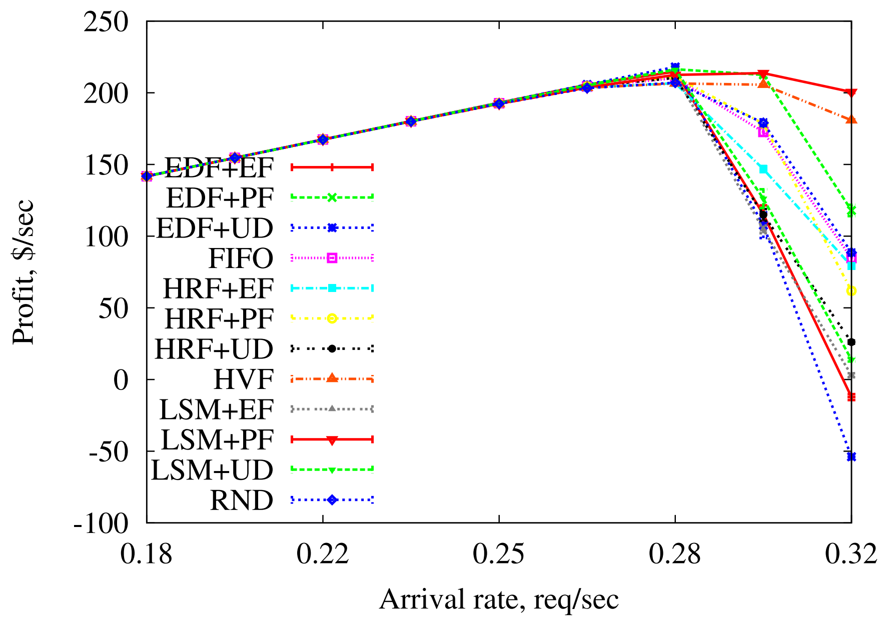

Figure 4.48: Profit. Medium business. Poisson arrivals. Uncorrelated SLAs.

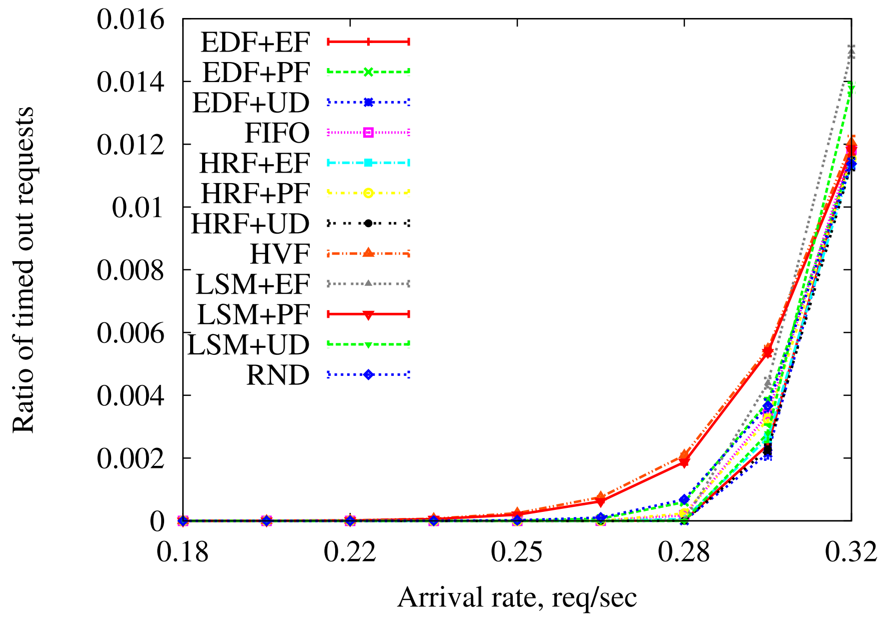

Figure 4.49: Ratio of requests which have timed out. Medium business. Poisson arrivals. Uncorrelated SLAs.

during the high load, the profit drop is much more substantial as opposed to the Poisson case.

LSM+* [Figure 4.51] in contrast to EDF+* [Figure 4.50] is able to avoid such a drastic drop of the profit. However, in contrast to the corresponding Poisson case, the maximum value of the profit (0.25-0.28 req/sec) is somewhat lower. Besides that, EF demonstrates the results stronger than UD, while PF is superior both to EF and UD.

According to Figure 4.52, the difference in traffic has a rather dramatic effect on the performance of the HRF+* policies. Apart from that, HRF also appears to be not the best choice in terms of the local scheduling policy, as in this case when HRF combined with PF, it fails to provide more or less consistent results. In this case HRF+PF shows the worst profit, while the best policy is HRF+UD. Such a behaviour can be attributed to the inability of HRF to meet deadlines, when used with a single service.

All of the proposed policies are compared in Figure 4.53. LSM+PF exhibits the best profit in the high load case, meanwhile HVF demonstrates the results of the same quality. Another interesting observation, is that despite the type of the traffic, in case of the correlated SLAs all policies appear to be better than FIFO or RND. Another group of policies showing strong results, in this scenario, are LSM+EF, EDF+PF, HRF+EF and HRF+UD. In terms of reliability, all the policies have very close results [Figure 4.54].

4.6.4 Non-Poisson Arrivals. Uncorrelated SLA

In this scenario, the request interarrival times were distributed according to a log-normal distribution with the SCV of 10.0. The clients contracted the composite services using the same agreements as in the Subsection 4.6.2.

The effect of using the proposed global scheduling policies with EDF is presented in Figure 4.55. In this case, PF demonstrates the best results most of the times, while, in contrast to Poisson case, the difference is much smaller. Generally speaking, the change in the arrival process from Poisson to heavy-tailed traffic has the same effect on the system with correlated SLAs. Therefore, it can be concluded, that the increase of the burstiness in the request arrival process has a negative effect of the efficiency of the EDF+PF policy.

At the same time, LSM+PF, is less susceptible to the difference in the request arrival pro-

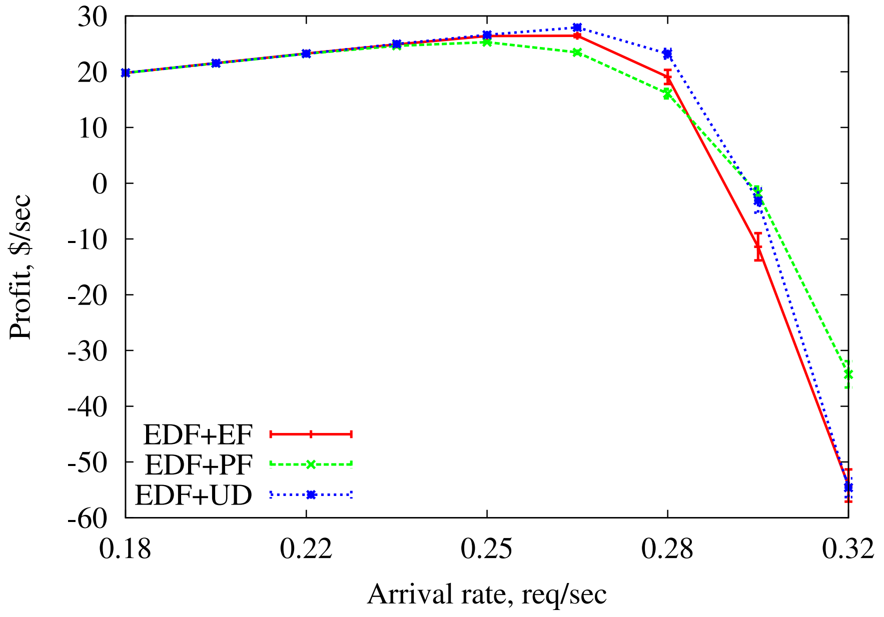

Figure 4.50: Profit. Medium business. Non-Poisson arrivals. Correlated SLAs. EDF+*.

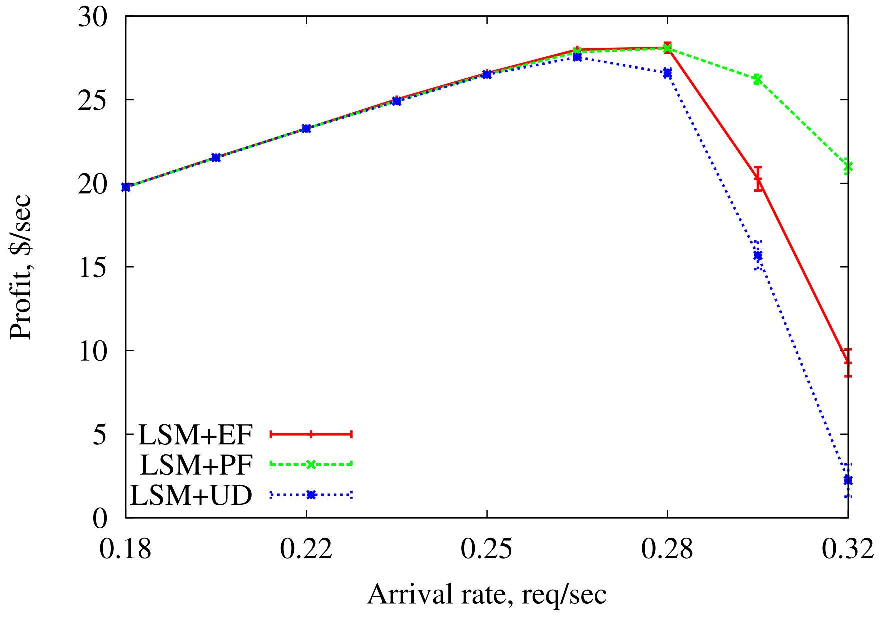

Figure 4.51: Profit. Medium business. Non-Poisson arrivals. Correlated SLA. LSM+*.

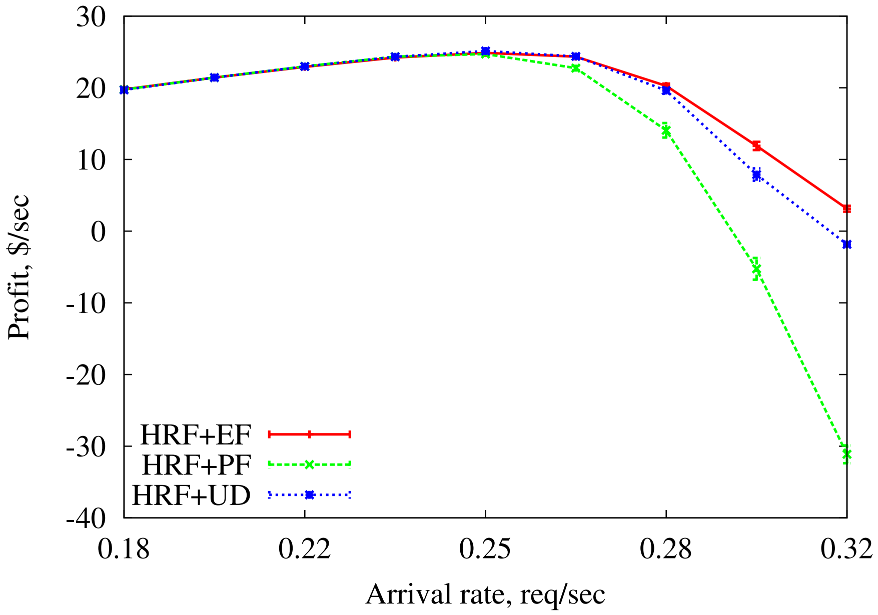

Figure 4.52: Profit. Medium business. Non-Poisson arrivals. Correlated SLAs. HRF+*.

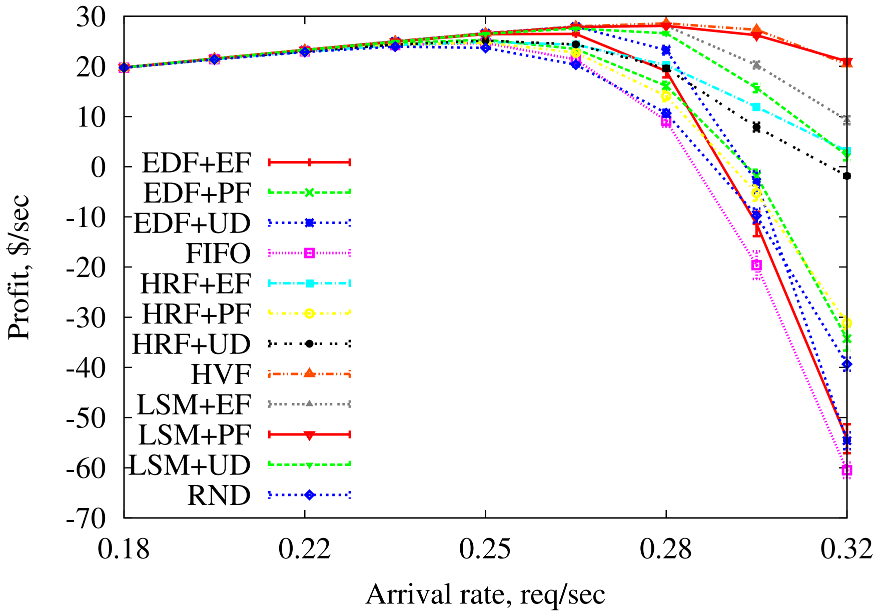

Figure 4.53: Profit. Medium business. Non-Poisson arrivals. Correlated SLAs.

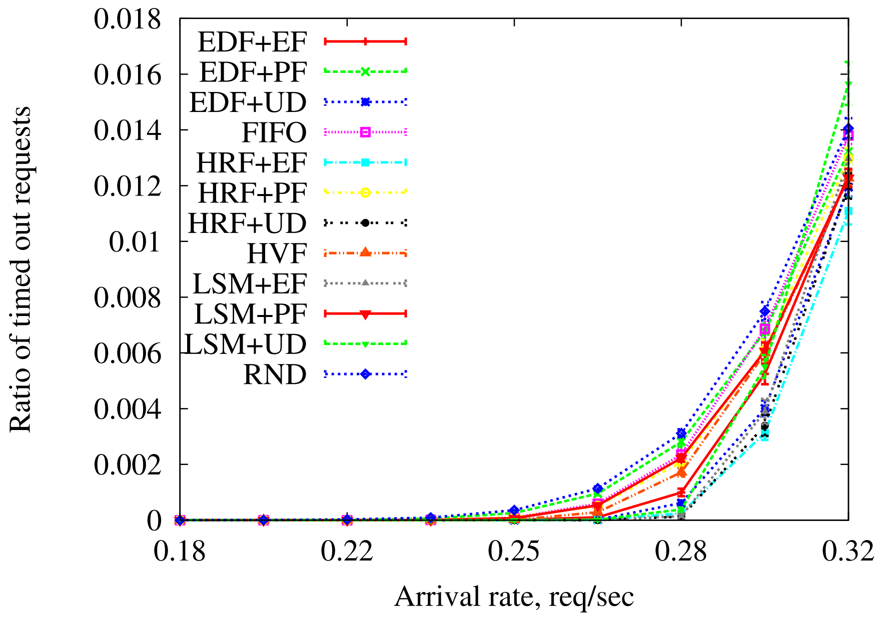

Figure 4.54: Ratio of requests which have timed out. Medium business. Non-Poisson arrivals. Correlated SLAs.

cess, as the policy still exhibits the results consistent with case two [Figure 4.56]. Moreover, same as in the second case, UD is slightly superior to ED, yet both of them are drastically outperformed by PF.

Figure 4.57 contains the results by HRF+* policies. Due to HRF's inability of correctly meeting the deadlines, shown in the previous subsection, all three policies demonstrated a substantial drop in the profit. Surprisingly, the worst results are obtained by using PF, while the UD is superior to its competitors (HRF+PF and HRF+EF) during the high load phases (0.28-0.32 req/sec).

The profits by all policies are presented in Figure 4.58. LSM+PF allows gaining the profit higher that the rest of the policies, meanwhile HVF shows the second best result. Unfortunatelly, as the load appraches 0.32 req/sec (the maximum throughput of the system) the other policies exhibit a steep drop of the profit. HRF+EF and EDF+PF appear to be the next best policies, even though their performance is not significanly better than the one by FIFO and RND. The chart in Figure 4.59 indicates that in terms of the reliability of the system, there is no substantial difference between the policies.

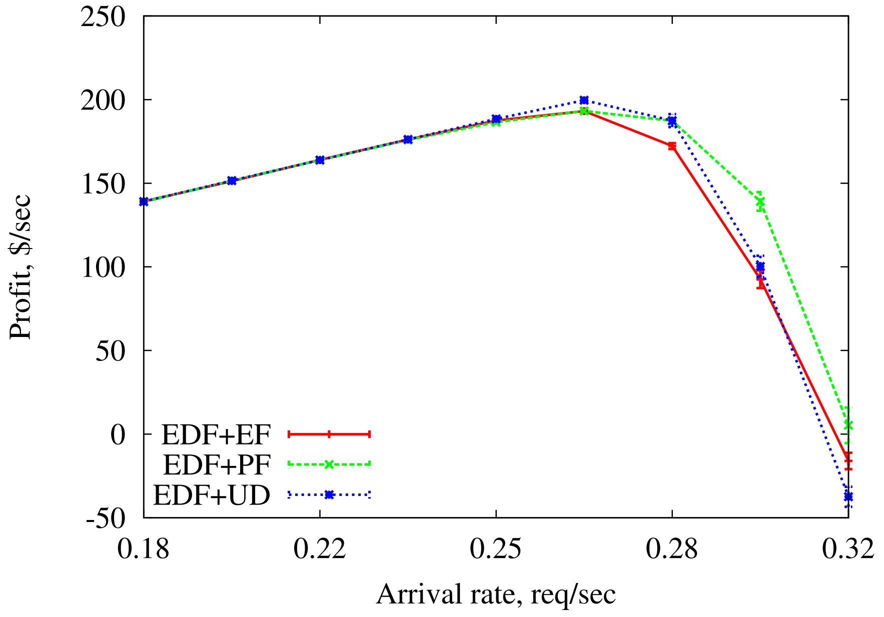

Figure 4.55: Profit. Medium business. Non-Poisson arrivals. Uncorrelated SLAs. EDF+*.

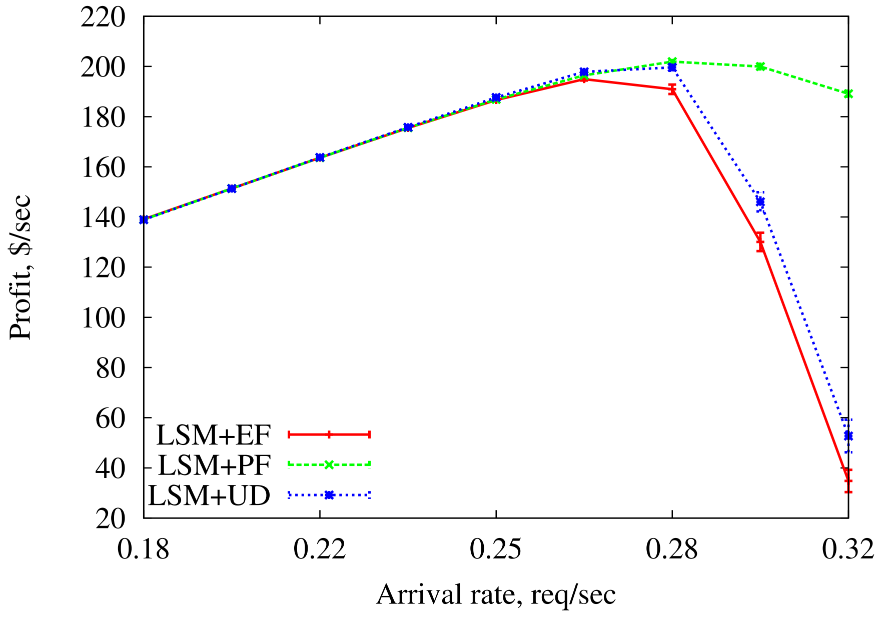

Figure 4.56: Profit. Medium business. Non-Poisson arrivals. Uncorrelated SLAs. LSM+*.

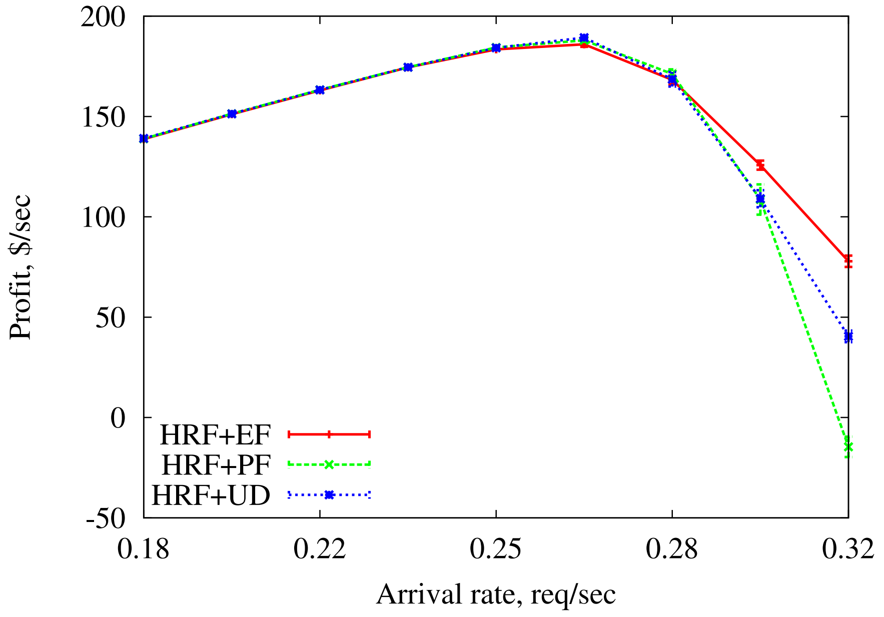

Figure 4.57: Profit. Medium business. Non-Poisson arrivals. Uncorrelated SLAs. HRF+*.

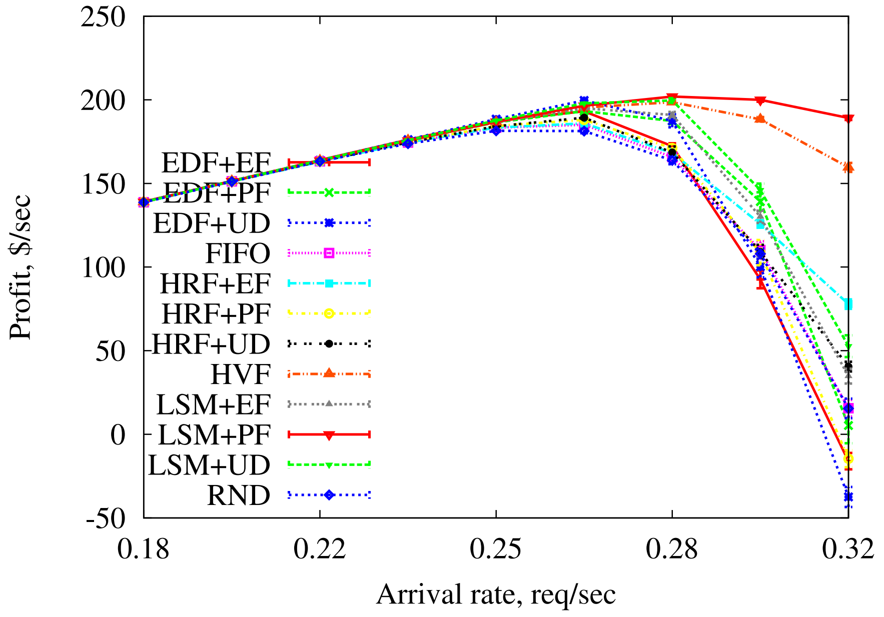

Figure 4.58: Profit. Medium business. Non-Poisson arrivals. Uncorrelated SLAs.

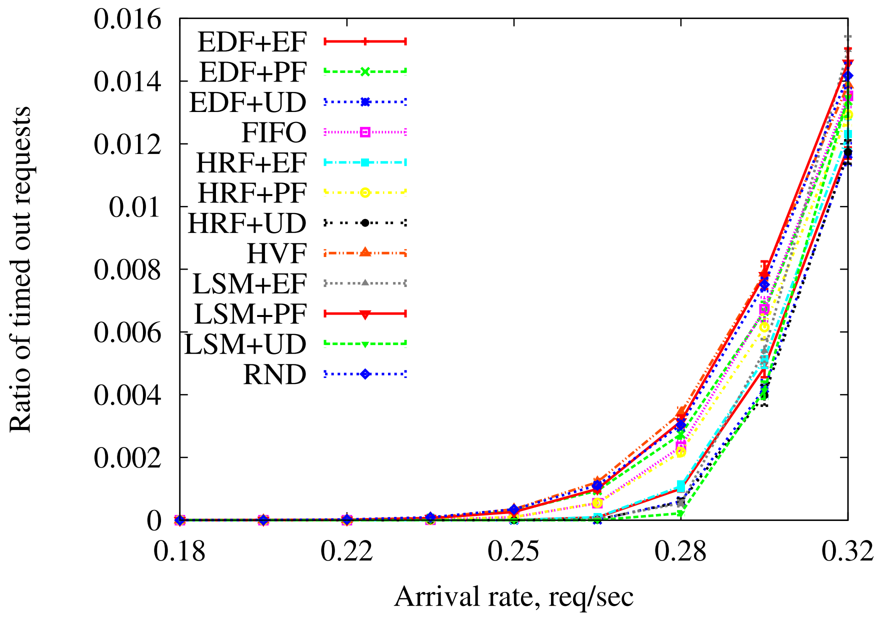

Figure 4.59: Ratio of requests which have timed out. Medium business Simulation. Non-Poisson arrivals. Uncorrelated SLAs.

4.6.5 Summary

The proposed global scheduling policies (EF, PF and UD) combined with the proposed local scheduling policies (LSM, EDF, HRF, HVF) were evaluated in a setting containing 100 atomic services and 40 workflows. The experiments were conducted in two dimensions, where the first dimension characterizes the variability of the traffic (Poisson or non-Poisson bursty traffic). Meanwhile, the second dimension was the type of SLAs. In the SLAs of the first type, the prices (penalties and hard penalties) of requests depend on the deadlines (hard deadlines). Meanwhile, in the second type of SLAs the deadlines and the prices are independent. The composite services were exposed to an increasing load and the generated profit was used as the main metric. The effects of the combining the scheduling policies are the following.

EDF+PF shows results persistently better than EDF+PF and EDF+UD, however the difference declines with an increase of the variability of the traffic. Moreover, the profit of EDF+PF is not affected by the type of SLAs. At the same time, EDF+PF and EDF+UD show very close results in all four scenarios.

As for LSM+*, LSM+PF exhibits the consistent supremacy in all four scenarios. The advantages of using this policy over LSM+EF or LSM+UD tend to increase with the switch from correlated SLAs to uncorrelated ones. Meanwhile, the traffic has no substantial effect on the profit. Moreover, the difference between LSM+EF and LSM+UD is less when those policies are used with the uncorrelated SLAs.

The type of traffic has a strong impact on the performance of HRF+PF, as its performance declines with the increase of the traffic variability. Nonetheless, there is no tangible difference in its behaviour with respect to the type of SLAs.

With respect to the global policies, LSM+PF is the absolute best policy, while HRF+PF is susceptible to the type of traffic. EDF+PF shows strong results across all four scenarios, nonetheless is still worse than LSM+PD or HVF. EF, in contrast to PF, does not exhibit such strong results in all cases. Moreover, in EDF+EF is very often among the worst performing policies, same as LSM+EF. However, when combined with HRF, in three out of four scenarios, it is better than HRF+UD or HRF+PF. As for UD, in the case of EDF, its outcomes are very similar to the ones produced by EDF+EF. If UD is used with LSM, the situation slightly improves, as in most of the scenarios, it outperforms LSM+EF. Meanwhile, HRF+UD is second best among HR+* policies only in two out of the four scenarios.

4.7 Summary

This section contains the description of the experiments conducted in order to evaluate the solution proposed in Chapter 3. The experiments were conducted using a simulator validated to experiments with a real-world Web Service.

The first section evaluates the proposed scheduling methods with respect to their complexity. The empirical study demonstrated that modern hardware allows the highest scheduling complexity of $O(N^2)$ if the waiting queue does not exceed 500 elements. In this case the scheduling overhead is less than one millisecond.

A series of experiments were conducted using a real-world service in order to observe the effect of the MPL on the resource utilization, response time, throughput and other metrics. A Web Service used by a local company was chosen as a subject of study.

Scheduling proxy, implementing a Static Priority scheduling policy (HVF), was used in combination with the aforementioned service. The experiments showed that HVF allows minimizing the number of SLA violations for more valuable customers.

Data from the previous experiment was employed for validating a Web Service model. The model was implemented by means of an event-driven simulator. Model validation was performed against response time, throughput, number of SLA violations, profits, revenues and penalties.

A comparison of the local scheduling policies (LSM, EDF, HVF and HRF) was conducted using the aforementioned model. In the simulation, the service was exposed to an increasing load of two types, a Poisson traffic and heavy-tailed traffic with the squared coefficient of variation of 10.0. Besides that, the policies were compared in scenarios where deadlines were inversely proportional to the paid price (correlated SLAs), as well as in scenarios where those two parameters were independent (uncorrelated SLAs). LSM showed the most consistent strong results, while using HVF appeared to be the most advantageous in the case of correlated SLA, and EDF in the case of uncorrelated SLAs.

In the last series of experiments, the local scheduling policies (LSM, EDF, HVF and HRF) were paired with the global scheduling policies (EF, PF and UD). The setting comprised 100 services and 40 randomly generated workflows. The four different scenarios differed by the type of SLAs (correlated or uncorrelated) and by the type of traffic (Poisson or heavy-tailed). LSM+PF showed the strongest results regardless of the scenario, while HVF, which is the simplest policy to deploy, demonstrated the second best result.

Chapter 5
Conclusions

This chapter ends the dissertation with some concluding remarks, thesis contribution and a list of possible future directions of research in this area.

5.1 Concluding Remarks

This thesis addresses the problem of meeting Service Level Agreements (SLAs) by means of scheduling. In the first part of this work an overview of theoretical findings and their practical applications is presented. The second part of the thesis focuses on scheduling policies as a means for ensuring stipulated Service Level Agreements for composite services specified using workflow languages (e.g. BPEL4WS [5]). In this thesis it is proposed to perform scheduling in two steps, first using a global scheduling policy and then using a local scheduling policy. The global scheduling policy first analyses the corresponding Service Level Agreement and then it decomposes the QoS requirements for a worlkflow into QoS requirements for component services invocations. Meanwhile, local schedulers, responsible for meeting those QoS requirements, process component services requests according to their timeliness and value.

Four different local policies are analyzed, each having specific deployment constraints. Highest Value First (HVF) requires only the business value of requests and thus has the minimal deployment requirements. Deadline based policies, Earliest Deadline First (EDF) and Highest Ratio First (HRF) also require the information on the deadlines associated with requests. Finally, the most advanced policy is using Lawler's scheduling method (LSM), which also requires information regarding the size of each request.

As for the global scheduling policies Equal Fraction (EF) and Proportional Fraction (PF) were suggested. EF recognizes nested structures of the workflows and allocates equal relative deadlines to component service invocations in the Critical Path of a workflow. As a result, it prioritizes only the critical components of the workflow. Meanwhile, non-critical resources are released to other workflows. Apart from the workflow structure, PF also takes into account the average response time of specific component services when allocating the deadlines. Consequently, it is able to more accurately distribute the time required for processing a workflow. It is worth stressing that the proposed solution does not require human intervention in order to be maintained and can operate without a single point of control. Furthermore, the proposed solution does not demand alternation of legacy services and is platform/programming language independent.

A prototype of a scheduling proxy implementing the HVF policy was developed and tested with a real-world Web Service. The evaluation of the proposed global and local policies was conducted using an event-driven simulation calibrated to a real-world service. Experiments with varying types of SLAs and different types of traffic demonstrated the feasibility of the proposed approach. The best results were achieved by LSM+PF, while using HVF appeared to be more advantageous in scenarios where more expensive requests have shorter deadlines.

5.2 Thesis Contributions

The following summarizes the main contributions of this thesis.

- Two global scheduling policies Equal Fraction and Proportional Fraction are proposed. The policies enable enforcing Service Level Agreements which stipulate composite service response time guarantees. The policies were developed for composite services implemented using workflow languages, e.g. BPEL4WS [Chapter 3].

- The feasibility of the scheduling was confirmed by conducting experiments with a specially developed low-latency transparent proxy. The proxy introduces the minimal message relay overhead and was tested with the Static Priority scheduling policy on a real-world commercial Web Service [Section 4.3].

- The advantages of the proposed scheduling policies are demonstrated by means of a simulation in a setting mimicking a medium scale company. The simulator used for the experiments was validated by means of experiments with a commercial Web Service. The experiment were conducted with different types of traffic and different types of Service Level Agreements [Sections 4.5, 4.6].

5.3 Future Work

In the future it is planned to expand the current work in the following directions:

- Workflows relying on third-party services. At the moment all work has been done assuming having full control over the services. However, the market of third-party services is rapidly expanding and more organizations use composite services which rely on external services, e.g. credit card services. Such services often are offered without any SLA while workflows incorporating them can be exposed with response time guarantees. Being able to compensate the fluctuations of the response time from third-party services constitutes another direction for future development.

- Experiments with heavy-tailed service times. Currently, the services times were assumed to distributed according to a log-normal distribution with the squared coefficient of variation of 0.5. However, investigating how the distribution of the service times affects the efficiency of the proposed policies is another potential direction of the future work.

- Nested workflows. Currently, only workflows using atomic services are considered. However, in practice workflows can invoke other composite services, which in their turn may also rely on composite services. Addressing such scenarios will contribute to the applicability of the proposed solution.

- Other types of SLAs. The experiments presented in this thesis were conducted with workflows exposed via SLA where the penalty increases over time (weighted number of late jobs). However, other scenarios exist, for example penalty function can be expressed through the weighted lateness [Figure 5.1].

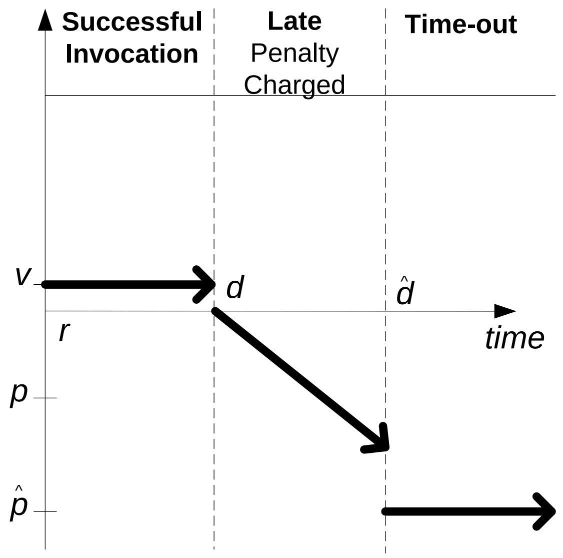

Figure 5.1: SLA model based on weighted lateness.

Bibliography

[1] *Apache Tomcat*, May 2011. http://tomcat.apache.org/. 2.2.1

[2] G. G. Abraham Silberschatz, Peter Baer Galvin. *Operating System Concepts*. John Wiley & Sons. Inc, 2005. 2.1

[3] R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint driven web service composition in METEOR-S. *Proceedings of the 2004 IEEE International Conference on Services Computing (SCC 2004)*, pages 23–30, 2004. 2.1

[4] R. J. Al-ali, O. F. Rana, D. W. Walker, S. Jha, and S. Sohail. G-QoSM: Grid service discovery using QoS properties. *Computing and Informatics Journal, Special Issue on Grid Computing*, 21:1–10, 2002. 2.1

[5] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services (BPEL4WS). Version 1.1*, May 2003. 1.1.4, 3.1, 5.1

[6] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. *Web Services Agreement Specification (WS-Agreement). Version 2005/09*, Sept. 2005. 1.1.3

[7] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Oceano-SLA based management of a computing utility. *Proceedings of 2001 IEEE/IFIP International Symposium on Integrated Network Management (IFIP/IEEE IM 2001)*, pages 855–868, 2001. IS:. 2.1

[8] D. Ardagna, M. Trubian, and L. Zhang. SLA based resource allocation policies in autonomic environments. *Journal of Parallel and Distributed Computing*, 67(3):259–270, 2007. 2.1

[9] J. Armstrong. Erlang - a survey of the language and its industrial applications. In *In Proceedings of the Symposium on Industrial Applications of Prolog (INAP96)*, 1996. 3.1.2

[10] J. Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, Swedish Institute of Computer Science, Stockholm, Sweden, 2003. 3.1.2

[11] K. Baker. Sequencing rules and due-date assignments in a job shop. *Management Science*, 30:1093–1104, 1994. 2.3.2, 2.3.3

[12] T. Baker. A stack-based resource allocation policy for realtime processes. *Proceedings of 11th Real-Time Systems Symposium (RTSS'90)*, pages 191–200, 1990. 2.2.3

[13] T. P. Baker. Stack-based scheduling of realtime processes. *The Journal of Real-Time Systems*, 3(1):67–100, 1991. 2.2.3

[14] P. Baptiste. Polynomial time algorithms for minimizing the weighted number of late jobs on a single machine with equal processing times. *Journal of Scheduling*, 2:245–252, 1999. 2.2.2

[15] P. Baptiste. Scheduling equal-length jobs on identical parallel machines. *Discrete Applied Mathematics*, 103(1):21–32, 2000. 2.2.2

[16] P. Baptiste, P. Brucker, S. Knust, and V. Timkovsky. Ten notes on equal-execution-time scheduling. *Discrete Optimization*, 2:111–127, 2004. 2.2.3

[17] P. Baptiste, M. Chrobak, C. Dürr, W. Jawor, and N. Vakhania. Preemptive scheduling of equal-length jobs to maximize weighted throughput. *Operations Research Letters*, 32(3)(3):258–264, 2004. 2.2.3

[18] S. Baruah, S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. *Journal of Real-Time Systems*, 4(2):125–144, 1992. 2.2.3

[19] J. C. R. Bennett and H. Zhang. WF^{2}Q: Worst-case fair weighted fair queueing. *Computer Networks*, pages 120–128, 1996. 2.2.4, 2.2.4

[20] Y. Bernet, P. Ford, R. Yavatkar, F. Baker, L. Zhang, M. Speer, R. Braden, B. Davie, J. Wroclawski, and E. Felstaine. A Framework for Integrated Services Operation over Diffserv Networks. RFC 2998 (Informational), Nov. 2000. 2.1

[21] J. Bertrand. The effect of workload dependent due-dates on job shop performance. *Management Science*, 29(7):799–816, 1983. 2.3.2

[22] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Handbook on Scheduling*. International Handbooks on Information Systems. Springer, 2007. 2.2.2

[23] D. Box. Four tenets of service orientation. Technical report, Microsoft, 2003. 1.1.1

[24] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol. RSVP 2205 (Informational), Sept. 1997. 2.1

[25] W. L. Brogan. *Modern control theory*. Prentice Hall, 1991. 2.2.1

[26] P. Brucker. *Scheduling Algorithms*. Springer-Verlag New York, Inc, 2001. 2.2.2, 2.2.2

[27] P. Brucker and S. Knust. Complexity results for single-machine problems with positive finish-start time-lags. *Journal of Computing*, 63:299–316, 1999. 2.3.1

[28] M. J. Buco, R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu. Utility computing SLA management based upon business objectives. *IBM Systems Journal*, 43(1):159–179, 2004. 2.1

[29] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, pages 90–98, 1995. 2.2.3, 2.2.3

[30] G. C. Buttazzo and J. Stankovic. RED: a robust earliest deadline scheduling algorithm. In *Proceedings of Third International Workshop on Responsive Computing Systems*, pages 90–98, 1993. 2.2.3

[31] G. C. Buttazzo and J. A. Stankovic. RED: Robust earliest deadline scheduling. Technical report, University of Massachusetts, Amherst, MA, USA, 1993. 2.2.3

[32] J. Cardoso. Stohastic workflow reduction algorithm. Technical report, LSDIS Lab, Department of Computer Science, Athens, Georgia, USA, 2002. 2.3.2

[33] J. Cardoso, J. Miller, A. Seth, and J. Arnold. Modelling quiality of service for workflows and web service processes. Technical Report 02-002 v2, LSDIS Lab, Department of Computer Science, University of Georgia, 2002. 2.3.2

[34] R. Clark. *Scheduling dependent real-time activities.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1990. 2.2.3

[35] R. W. Conway. *Theory of scheduling.* Addison-Wesley Pub. Co., Reading, MA, USA, 1967. 2.2.2, 2.3.2

[36] G. Cooper, L. DiPippo, L. Esibov, R. Ginis, R. Johnston, P. Kortman, P. Krupp, J. Mauer, M. Squadrito, B. Thurasignham, S. Wohlever, and V. Wolfe. Real-time CORBA development at MITRE, NRaD, TriPacific and URI. In *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, 1997. 2.2.3, 2.2.3

[37] T. Cucinotta, A. Mancina, G. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusina. A real-time service-oriented architecture for industrial automation. *IEEE Transactions on Industrial Informatics*, 5(3):267–277, 2009. 2.2.3, 2.4

[38] A. Dan, D. Davis, R. Kearney, A. Keller, R. P. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: WSLA-driven automated management. *IBM Systems Journal*, 43(1):136–158, 2004. 1.1.3, 2.1

[39] P. J. Denning. Thrashing: Its causes and prevention. In *Proceedings of AFIPS, Fall Joint Computer Conference*, 1968. 2.2.1

[40] Y. Diao, J. Hellerstein, and S. Parekh. Using fuzzy control to maximize profits in service level management. *IBM Systems Journal*, pages 403–420, 2002. 2.2.1

[41] X. Ding, X. Zhang, B. Jin, and T. Huang. A task-type aware transaction scheduling algorithm in J2EE. In *Proceedings OTM Conferences (2)*, volume 3761 of *Lecture Notes in Computer Science*, pages 1034–1045. Springer, 2005. 2.1, 2.2.3

[42] D. Dyachuk and R. Deters. The impact of transient loads on the performance of service ecologies. In *Proceedings of the 2007 Inagural IEEE International Conference on Digital Ecosystems and Technologies (DEST 2007)*, 2007. 2.2.1

[43] D. Dyachuk and R. Deters. Optimizing performance of web service providers. In *Proceedings of the IEEE International Conference on Advanced Information Networking and Applications (AINA-2007)*, pages 46–53, 2007. 2.2.2, 4.1

[44] D. Dyachuk and R. Deters. Service level agreement aware workflow scheduling. In *2007 IEEE International Conference on Services Computing (SCC 2007)*, pages 715–716. IEEE Computer Society, 2007. 2.2.2, 2.3.3, 3.1.1, 3

[45] D. Dyachuk and R. Deters. Transparent scheduling of web services. In *Proceedings of the 3rd International Conference on Web Information Systems and Technologies (WEBIST 2007)*, pages 112–119, 2007. 3.1.2, 3.4, 4.1

[46] D. Dyachuk and R. Deters. Using SLA context to ensure quality of service for composite services. In *Proceedings of the IEEE International Conference on Pervasive Services (ICPS 2007)*, pages 64–67, 2007. 2.2.1, 3

[47] D. Dyachuk and R. Deters. A solution to resource underutilization for web services hosted in the cloud. In *OTM '09 Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems*, volume 1, pages 567–584, 2009. 2.1, 2.4

[48] D. Dyachuk and M. Mazzucco. On allocation policies for power and performance. In *Proceedings of the 11th ACM/IEEE International Conference on Grid Computing (GRID 2010)*, pages 313–320, 2010. 4.4

[49] D. Dyachuk and R.Deters. Ensuring service level agreements for service workflows. In *2008 IEEE International Conference on Services Computing (SCC 2008)*, volume 2, pages 333–340, 2008. 2.2.2, 3.1.1

[50] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th international conference on World Wide Web (WWW'04)*, pages 276–286, 2004. 2.2.1

[51] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. Price. Grid service orchestration using the business process execution language (BPEL). *Journal of Grid Computing*, 3(3-4):283–304, 2005. 1.1.4

[52] A. Erradi and P. Maheshwari. wsBus: QoS-aware middleware for reliable web services interactions. In *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05)*, pages 634–639, 2005. 1.1.5

[53] D. Faggioli, M. Trimarchi, and F. Checconi. An implementation of the earliest deadline first algorithm in linux. In *Proceedings of the 2009 ACM symposium on Applied Computing (SAC '09)*, pages 1984–1989, 2009. 2.2.3, 2.4

[54] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Technical report, World Wide Web Consortium (W3C), 1999. 4.2

[55] R. T. Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, California, USA, 2000. 3.1

[56] S. French. *Sequencing and scheduling :an introduction to the mathematics of the job-shop.* E. Horwood Ltd., Chichester, West Sussex, 1982. 2.2.2, 2.2.2

[57] S. Frolund and J. Koisten. *QML: A Language for Quality of Service Specification.* HP Labs Technical Reports, Palo Alto, California, USA, hpl-98-10 edition, 1998. 1.1.3

[58] W. Gere. Heuristics in job shop scheduliung. *Management Science*, 13:167–190, 1966. 2.3.2

[59] C. D. Gill, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time CORBA scheduling service. *Real-Time Systems*, 20(2):117–154, 2001. 2.2.3

[60] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Capacity management and demand prediction for next generation data centers. In *Proceedings of 2007 IEEE International Conference on Web Services (ICWS 2007)*, pages 43–50, 2007. 2.1, 2.2.4

[61] V. Gordon, J.-M. Proth, and C. Chu. A survey of the state-of-the-art of common due date assignment and scheduling research. *European Journal of Operational Research*, 127(1):1–25, 2002. 2.3.2

[62] P. Goyal, H. M. Vin, and H. Chen. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *Conference proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '96)*, pages 157–168, 1996. 2.2.4, 2.2.4

[63] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, 1997. 2.2.4, 2.2.4

[64] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, pages 15:287–326, 1979. 2.2.2

[65] H.-U. Heiss and R. Wagner. Adaptive load control in transaction processing systems. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB '91)*, pages 47–54, 1991. 2.2.1

[66] Y. Hu, J. Wong, G. Iszlai, and M. Litoiu. Resource provisioning for cloud computing. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '09)*, pages 101–111, 2009. 2.2.4

[67] ISO. Quality vocabulary (ISO/IEC Standard 8402), 1986. 1.1.2

[68] D. Z. J. Adams, E. Balas. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34:391–401, 1988. 2.3.2

[69] J. Jackson. Scheduling in a production line to minimize maximum tardiness. Research report 43, University of California, Los Angleles, California, USA, 1955. 2.3.1

[70] E. Jensen, C. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'85)*, pages 112–122, 1985. 2.2.3

[71] S. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1:61–68, 1954. 2.3.1

[72] H. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):428–437, 1993. 2.3.3, 2.4, 3.2.2

[73] M. Karlsson, C. Karamanolis, and J. Chase. Controllable fair queuing for meeting performance goals. *Performance Evaluation*, 62(1-4):278–294, 2005. 2.2.4, 2.2.4

[74] R. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Plenum, 1972. 2.2.2

[75] A. Keller and H. Ludwig. The WSLA Framework: Specifying and monitoring service level agreements for Web Services. *IBM Research Report*, May 2002. 1.1.3

[76] E. Kim and Y. Lee. *OASIS Web Services Quality Model*. Organization for the Advancement of Structured Information Standards, Semptember 2005. 1.1.2

[77] C. M. K.Jeffay, D.F. Stanat. On non-preemptive scheduling of period and sporadic tasks. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'91)*, pages 129–139, 1991. 2.2.3

[78] L. Kleinrock. A conservation law for a wide class of queueing disciplines. *Naval Research Logistics Quarterly*, 12:181–192, 1965. 2.2.4

[79] G. Koren and D.Shasha. D-over: An optimal on-line scheduling alogorithm for overloaded real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'92)*, pages 290–299, 1992. 2.2.3

[80] D. D. Lamanna, J. Skene, and W. Emmerich. SLAng: a language for defining service level agreements. In *Proceedings the Ninenth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003)*, pages 100–106, 2003. 1.1.3

[81] E. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, 19:544–546, 1973. 2.2.2, 3.3, 4

[82] E. Lawler. A "pseudopolynomial" algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics*, 1:331–342, 1977. 2.2.2

[83] E. Lawler. A dynamic programming algorithm for preemptive scheduling of a single machine to minimize the number of late jobs. *Annals of Operation Research*, 26(1-4):125–133, 1990. 2.2.3

[84] E. Lawler and J. Moore. A functional equation and its application to resource allocation and sequencing problems. *Management Science*, 16:77–84, 1969. 2.2.2, 2.2.4

[85] J. Lenstra, A. G. R. Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977. 2.2.2, 2.2.4

[86] J. Y.-T. Leung, editor. *Handbook of Scheduling: Alogorithms, Models, and Performance Analysis*. Chapman & Hall/CRC, 2004. 2.2.2, 2.2.3, 2.2.3

[87] J. Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Inf. Process. Lett.*, 11(3):115–118, 1980. 2.2.3

[88] P. Li. Time/utility function decomposition techniques for utility accrual scheduling algorithms in real-time distributed systems. *IEEE Transactions on Computers*, 54(9):1138–1153, 2005. 2.2.3, 2.3.3

[89] P. Li, H. Wu, B. Ravindran, and E. D. Jensen. A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints. *IEEE Transactions on Computers*, 55(4):454–469, 2006. 2.2.3

[90] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1):46–61, 1973. 2.2.3, 2.2.3

[91] S. Liu, G. Quan, and S. Ren. On-Line scheduling of Real-Time services for cloud computing. In *Proceedings of the IEEE Congress on Services (SERVICES 2010)*, pages 459–464, 2010. 2.2.4, 2.4

[92] Z. Liu, M. S. Squillante, and J. L. Wolf. On maximizing service-level-agreement profits. In *EC '01: Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 213–223, 2001. 2.2.4, 2.2.4

[93] C. D. Locke. *Best-effort decision-making for real-time scheduling.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1986. 2.2.3, 2.2.3, 3

[94] A. Lodde, A. Schlechter, P. Bauler, N. Biri, and F. Feltz. Feedback controlled quality of service enforcement for service oriented architectures. In *Proceeding of the IEEE International Conference on Services Computing 2010, (SCC 2010)*, pages 354–361, 2010. 2.2.4, 2.4

[95] H. Ludwig. Web Services QoS: External SLAs and internal policies - or: How do we deliver what we promise? In *Proceeding of the 4th IEEE International Conference on Web Information System Engineering (WISE'03)*, pages 115–120, 2003. 2.1

[96] A. Mani and A. Nagarajan. Understanding quality of service for Web Services. Online, Jan 2002. http://www.ibm.com/developerworks/webservices/library/ws-quality/index.html. 1.1.2

[97] M. Mazzucco, D. Dyachuk, and M. Dikaiakos. Profit-aware server allocation for green internet services. In *Proceedings of the 18th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2010)*, pages 277–284, 2010. 4.4

[98] D. W. McCoy and Y. V. Natis. Service-oriented architecture: Mainstream straight ahead. Technical report, Gartner, Inc., 2003. 1.1.1

[99] S. McCready. There is more than one kind of workflow software. *Computer World*, 2:86–90, 1992. 1.1.4

[100] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 535, 2004. 2.1, 2.2.3

[101] D. A. Menascé, V. A. F. Almeida, R. Fonseca, and M. A. Mendes. Business-oriented resource management policies for e-commerce servers. *Performance Evaluation*, 42(2-3):223–239, 2000. 2.2.4

[102] J. Moore. An n job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15:102–109, 1968. 2.2.4

[103] G. C. Naccache, Henri; Gannod. A self-healing web server using differentiated services. *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC 2006)*, pages 203–214, 2006. 2.2.2

[104] G. C. Naccache, Henri; Gannod. A self-healing framework for Web Services. *Proceedings of the IEEE International Conference on Web Services, 2007 (ICWS 2007)*, pages 398–345, 2007. 2.2.2

[105] J. O'Hara. Toward a commodity enterprise middleware. *ACM Queue*, 5(4):48–55, 2007. 1.1.5

[106] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSa - adaptive quality of service architecture. *Software Practice and Experience*, 39:1–31, January 2009. 2.2.3

[107] S. Panwalkar. A survey of scheduling rules. *Operation Research*, 25:45–61, 1977. 2.3.2

[108] M. P. Papazoglou and B. Kratz. Web services technology in support of business transactions. *Service Oriented Computing and Applications*, 1(1):51–63, 2007. 1.1.5

[109] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Kramer. Service-oriented computing: A research roadmap. In F. Cubera, B. J. Kramer, and M. P. Papazoglou, editors, *Dagstuhl Seminar Proceedings*, pages 1–29, 2006. 1.1.4, 1.1.5

[110] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *Journal IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993. 2.2.4, 2.2.4

[111] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'01)*, 2001. 2.2.1

[112] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Prenctice Hall, 2001. 3.2.2

[113] N. Raman and F. Talbot. The job shop tardiness problem: A decomposition approach. *European Journal of Operational Research*, 69:187–199, 1993. 2.3.2

[114] S. Ran. A model for web services discovery with QoS. *ACM SIGecom Exchanges*, 4(1):1–10, 2003. 1.1.2

[115] A. Sahai, A. Durante, and V. Machiraju. Towards automated sla management for web services. Research report hpl-2001-310 (r.1), Hewlett-Packard Laboratories, 2002. 1.1.3

[116] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman. How to determine a good Multi-Programming level for external scheduling. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 60–72, 2006. 2.2.1, 2.2.3, 4.2, 4.4

[117] R. W. Schulte and Y. V. Natis. Service Oriented Architecture. SSA Research Note SPA-401-068, Gartner Group, 1996. 1.1.1

[118] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2):101–155, November 2004. 2

[119] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990. 2.2.3

[120] A. Sharma, H. Adarkar, and S. Sengupta. Managing QoS through prioritization in Web Services. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering Workshops (WISEW'03)*, pages 140–148, 2003. 2.2.4

[121] P. Siddhartha, R. Ganesan, and S. Sengupta. Smartware - a management infrastructure for web services. In *Proceedings of the 1st Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI-2003)*, pages 42–49, 2003. 2.2.4

[122] B. Simmons. Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *Siam Journal on Computing*, 12(2):294–299, 1983. 2.2.2

[123] B. Simons. A fast algorithm for single processor scheduling. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS '78)*, pages 246–252, 1978. 2.2.2

[124] A. Singhai, A. Sane, and R. H. Campbell. Reflective ORBs: Supporting robust, time-critical distribution. In *Proceedings of the Workshops on Object-Oriented Technology, 1997 (ECOOP '97)*, pages 55–61, 1997. 2.2.3

[125] L. Song and G. A. Marin. Generating realistic network traffic for security experiments. In *Proceedings of the IEEE SoutheastCon, 2004*, pages 200–207, 2004. 4.3, 4.5

[126] V. Tanaev, Y. Sotskov, and V. Strusevich. *Scheduling theory. Multi-stage systems.* Kluwer Academic Publishers Group, 1994. 2.3.1

[127] V. Tosic, H. Lutfiyya, and Y. Tang. Web Service offerings language (WSOL) support for context management of mobile/embedded XML Web Services. In *Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW '06)*, pages 156–166, 2006. 1.1.3

[128] V. Tosic, B. Pagurek, K. Patel, B. Esfandiari, and W. Ma. Management applications of the web service offerings language (WSOL). *Information Systems*, 30(7):564–586, 2005. 1.1.3

[129] Transaction Processing Performance Council. *TPC-App - Application Server and Web Service Benchmark*. http://www.tpc.org/tpc-app/. 2.3, 4.1

[130] W. Tsai, Y. Lee, Z. Cao, Y. Chen, and B. Xiao. RTSOA: Real-Time Service-Oriented architecture. In *Proceedings of the Second IEEE International Workshop on Service-Oriented System Engineering, 2006 (SOSE '06)*, pages 49–56, 2006. 2.4

[131] P. Uthaisombut. Generalization of EDF and LLF: Identifying all optimal online algorithms for minimizing maximum. *Algorithmica*, 50:312–328, 2006. 2.2.3

[132] W. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barroso. Workflow patterns. *Distributed Parallel Databases*, 14(1):5–51, 2003. 1.1.4, 2.3.2

[133] T. Vercauteren, P. Aggarwal, X. Wang, and T. Li. Hierarchical forecasting of web server workload using sequential Monte Carlo training. *IEEE Transactions on Signal Processing*, 55:1286–1297, 2007. 2.2.4, 2.3.4

[134] S. Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89, 2006. 1.1.5

[135] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems (HotOS'03)*, volume 9, pages 4–12, 2003. 3.1.2

[136] C. A. Waldspurger and E. Weihl. Stride scheduling: Deterministic proportional- share resource management. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995. 2.2.4

[137] M. D. Welsh. *An architecture for highly concurrent, well-conditioned internet services.* PhD thesis, University of California, Berkeley, California, USA, 2002. 3.1.2

[138] W. Whitt. Heavy traffic approximations for service systems with blocking. *AT&T Bell Laboratories Technical Journal*, 63:689–708, 1984. 2.2.4

[139] Y. Yu, S. Ren, N. Chen, and X. Wang. Profit and penalty aware (PP-aware) scheduling for tasks with variable task execution time. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010)*, pages 334–339, 2010. 2.2.4, 2.4

[140] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *Proceedings of the 12th International Conference on World Wide Web (WWW '03)*, pages 411–421, 2003. 2.1

[141] C. Zhang, R. N. Chang, C.-S. Perng, E. So, C. Tang, and T. Tao. QoS-aware optimization of composite-service fulfillment policy. *Proceeding of the IEEE International Conference on Services Computing, 2007 (SCC 2007)*, pages 11–19, 2007. 2.3.4, 2.4

[142] L. Zhang and D. Ardagna. SLA based profit optimization in autonomic computing systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC '04)*, pages 173–182, 2004. 2.2.4

[143] Z.-L. Zhang, D. F. Towsley, and J. F. Kurose. Statistical analysis of generalized processor sharing scheduling discipline. *IEEE Journal on Selected Areas in Communications*, 13(6):1071–1080, 1995. 2.2.4, 2.2.4

[144] X. Zhou, J. Wei, and C.-Z. Xu. Quality-of-service differentiation on the internet: a taxonomy. *Journal of Network and Computer Applications*, 30(1):354–383, 2007. 2.1