Towards Achieving Execution Time Predictability in Web Services Middleware

A thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

Wasala Appuhamilage Vidura Saminda Gamini Abhaya
B.Sc. (Hons.)

School of Computer Science and Information Technology,
College of Science, Engineering, and Health,
RMIT University

July, 2012

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Wasala Appuhamilage Vidura Saminda Gamini Abhaya
School of Computer Science and Information Technology
RMIT University
July, 2012

Acknowledgments

I would like to thank my supervisors Professor Zahir Tari and Associate Professor Peter Bertok for all their support and guidance throughout the past few years. Their continuous encouragement and feedback was greatly responsible for the success of this research. I'm also very grateful to Professor Panlop Zeephongsekul for being so selfless in helping me with a part of my research. I cannot thank him enough for taking time off his schedule to help me out with many of my questions, going through my work and giving some great feedback, despite not being directly involved with my research.

I am thankful to the School of Computer Science and I.T. at RMIT for the opportunity given to do a research degree and their financial support throughout the degree. I would also like to thank Dr. Falk Scholer, Dr. John Thangarajah and Associate Professor Michael Winikoff who were research coordinators throughout my candidature, for their help and guidance. I would like to thank Mrs. Beti Stojkovski and Miss. Dora Drakopoulous, the research administration officers at the start and end of my candidatures for all their support. I am thankful to the administrative staff at the school of Computer Science and I.T., especially to Ms. Dora Poulakis and Ms. Averil Newman.

I extend a special thanks to Don Gingrich for his friendship and all his help in finding me the required hardware and being so gracious in allowing me use the lab servers for my experiments. I thank all my colleagues at the Distributed Systems and Networking discipline for their support throughout the years. I have met some great people there and made lifelong friends. A special thanks goes to Malith Jayasinghe for all the interesting discussions throughout the year and to Ermyas Abebe and Anshuman Mukherjee for being such great mates. I am also thankful to Sunidhi Bhalla, Nalaka Gooneratne and Sarvanan Dayalan for all their support and friendship throughout the years. You all have helped me in many ways than you know and I am truly grateful for it.

All of this would not have been possible if not for the constant love and support I received from my family. Thank you for your understanding and tolerating with what seemed to be a never ending research. A special thanks to my mother who visited us in the last four months of my candidature, to help out with little Vishmi. Your presence made the biggest difference as I was able to concentrate on finishing up my research. I also thank my in-laws for their love and support throughout this period and especially visiting us and helping out with the birth of our little Vishmi. My sister-in-

law and brother-in-law need to be mentioned specially for all their support during this time period.

Little Vishmi, our daughter was my source of happiness away from work. Her beautiful smile kept me motivated to wrap things up quickly amidst some busy times and I'm thankful to her for that. Finally, this research would not have been possible if not for the love and support I received from my beautiful wife Piyumi. She has been my pillar of strength throughout the years and was a constant source of encouragement that kept me going through difficult times. I love you so much and am eternally grateful for all of that you made possible.

To ***Piyumi***,

who selflessly gave up on so many of her dreams to make one of mine a reality...

..and to our little ***Vishmi***,

who made this research work much more enjoyable.

Credits

Portions of the material in this thesis have previously appeared in the following publications:

- Gamini Abhaya V., Tari Z. and Bertok P.: Achieving Predictability and Service Differentiation in Web Services. In: ICSOC-ServiceWave 09: Proceedings of the 7th International Conference on Service-Oriented Computing, Stockholm, Sweden, pp. 364-372. Springer (2009).
- Gamini Abhaya V.,Tari, Z. and Bertok, P.:Using Real-Time Scheduling Principles in Web Service Clusters to Achieve Predictability of Service Execution. In: Proceedings of 8th International Conference on Service-Oriented Computing: ICSOC 2010, San Francisco, CA, USA, pp. 197-212. Springer (2010).
- Gamini Abhaya V., Tari Z. and Bertok P.: Building Web Services Middleware with Predictable Service Execution. In: Proceedings of 11th International Conference on Web Information Systems Engineering: WISE 2010, Hong Kong, pp. 23-37. Springer (2010).
- Gamini Abhaya V., Tari Z. and Bertok P.: Building Web Services Middleware with Predictable Service Execution. In: World Wide Web Journal: Special Issue on Web Information Systems Engineering. pp 1-60. Springer (2012).
- Gamini Abhaya V. Tari Z. Zeephongsekul P. and Zomaya A.: Waiting Time Analysis for a Multi-class Preemptive $M/G/1/./EDF$ queue. In: Journal of Parallel and Distributed Computing. Elsevier (To Appear).

The thesis was written using the `TeXmaker` editor on Ubuntu GNU/Linux, and typeset using the LaTeX 2_ε document preparation system.
All trademarks are the property of their respective owners.

Contents

List of Figures

List of Tables

Abstract

This thesis proposes analytical models, algorithms and software engineering techniques that enable web services to achieve execution times that are predictable and consistent. Their growth as the preferred middleware for communication among distributed systems in multitude of networks, demand better execution time performance.

Web services middleware are typically designed optimised for throughput. They accept every request receives and make no differentiation in processing them. Many use the *thread-pool* pattern to execute requests in parallel using processor sharing scheme. Clusters hosting web services dispatch requests to only balance out the load among the executors. Such optimisations for throughput work out negatively on the predictability of execution. Processor sharing results in the increase of execution time with the number of requests being processed in parallel. As a result, it becomes impossible to predict or control the execution of any web service request.

Existing works on execution time quality of service fail to address the need for predictability in web service execution. Some of the work achieve a level of differentiated processing, but fails to consider predictability as the main quality attribute. Some attempts try to give a probabilistic guarantee of service levels as outlined in service agreements. Nevertheless, from a predictability point-of-view they do not happen repeatedly and consistently. A few attempts manage to achieve predictable execution times, however only in closed systems where request properties are known at design time. Web services operate on the Internet, which is an open environment where request properties are relatively unknown.

In this thesis we investigate the problem of achieving predictable times in web service

executions. We introduce the notion of a *processing deadline* for service execution, that the web services engine must adhere to in completing the request in a repeatable and a consistent manner. Reaching such execution deadlines by the services engine is made possible by three main features. First a deadline based scheduling algorithm introduced, ensures the processing deadlines are followed. A laxity based analytical model and an admission control algorithm it is based on, selects requests for execution, resulting in a wider range of laxities to enable more requests with overlapping executions to be scheduled together. Finally, a real-time scheduler component introduced in to the server uses a priority model to schedule the execution of requests by controlling the execution of individual worker threads in custom-made thread pools. Predictability of execution in cluster based deployments is further facilitated by four dispatching algorithms that consider the request deadlines and laxity property in the dispatching process. All of them follow the request selection process that maximises the range of laxities at each cluster server, while the RT-Laxity algorithm further ensures widening of the range by keeping track of the last two laxities assigned to each server and avoiding their successive assignment to the same servers. The implementation of these features are further supported by development platforms and operating systems with real-time features. A performance model derived for a similar system approximates the waiting time where requests with smaller deadlines (considered to be higher priority) experience smaller waiting times than requests with longer deadlines.

All these techniques are implemented in popular web services middleware as standalone and cluster-based configurations. They are evaluated against their unmodified versions and other algorithms such as round-robin and class based dispatching, to measure the predictability gain achieved through the enhancements. Empirical evidence indicate that our enhancements enable the middleware products to achieve more than 90% of the deadlines, while accepting at least 20% of the requests in high traffic conditions. The enhancements additionally prevent the middleware from reaching overloaded conditions in heavy traffic, and maintain comparable throughput rates to the unmodified versions of the middleware. Analytical and simulation results for the performance model confirms that deadline based preemptive scheduling results in a better balance of waiting times where high priority requests experience lower waiting times and lower priority requests are not over-starved compared to other techniques such as static priority ordering, First-Come-First-Served, Round-Robin and non-preemptive deadline based scheduling.

Achieving such a level of predictability within web services middleware opens up var-

ious new application areas to the use of web services. Applications such as industrial control systems, avionics systems, medical equipment control systems, capital market trading systems and robotics mandate such stringent predictability in execution. Such systems with real-time requirements would greatly benefit from the inherent advantages Web services bring in, such as open protocols that would free component designers and developers from using proprietary methods of communication in such systems. This research hopes to open up such new application avenues to the use of Web services as a viable middleware platform.

Chapter 1

Introduction

The Internet has evolved beyond its traditional role of being a large collection of information sources into the true form of a large collection of distributed systems. It has been transformed from being user-centric to application-centric and is fast moving towards being a completely automated web [Cerami and St Laurent, 2002], where applications automatically discover and communicate with each other without any user intervention. This paradigm shift was possible only with the advent of web services technology, currently considered to be the *de-facto* standard for communication in distributed systems [Gartner and Forrester, 2003]. At the inception of web services, other distributed communication technologies such as Common Request Broker Architecture (CORBA) [Vinoski, 1993], Microsoft Distributed Component Object Model (DCOM) [Microsoft Corporation, 2012] and Java Remote Method Invocation (RMI) [Oracle, 2010] were widely in use. The quick adoption of Web services can be attributed to their self-describing, loosely-coupled, platform-independent nature and their minimal requirements for operation. The use of standard protocols such as Hypertext Transfer Protocol (HTTP) for transport, by Simple Object Access Protocol (SOAP) which is an Extensible Markup Language (XML) based data format, required no additional software or changes to networking infrastructure, such as routers and firewalls for web services communications to be supported.

Figure 1.1 illustrates an example of booking a holiday through a travel website on the Internet. Although the client interaction is limited to the travel website, it uses a reservation engine hosted somewhere on the Internet and exposed using web services. The reservation engine in turn communicates with several other systems such as airline

reservation, hotel reservation, car-rental and insurance where all of them are accessed through web services. Given a single booking done through the travel website, there maybe hundreds of web service invocations that take place to complete the operation, among the systems involved. Therefore, the execution time performance of each service invocations is critical for the completion of the task on-time, at the reservation engine and in turn the travel website.

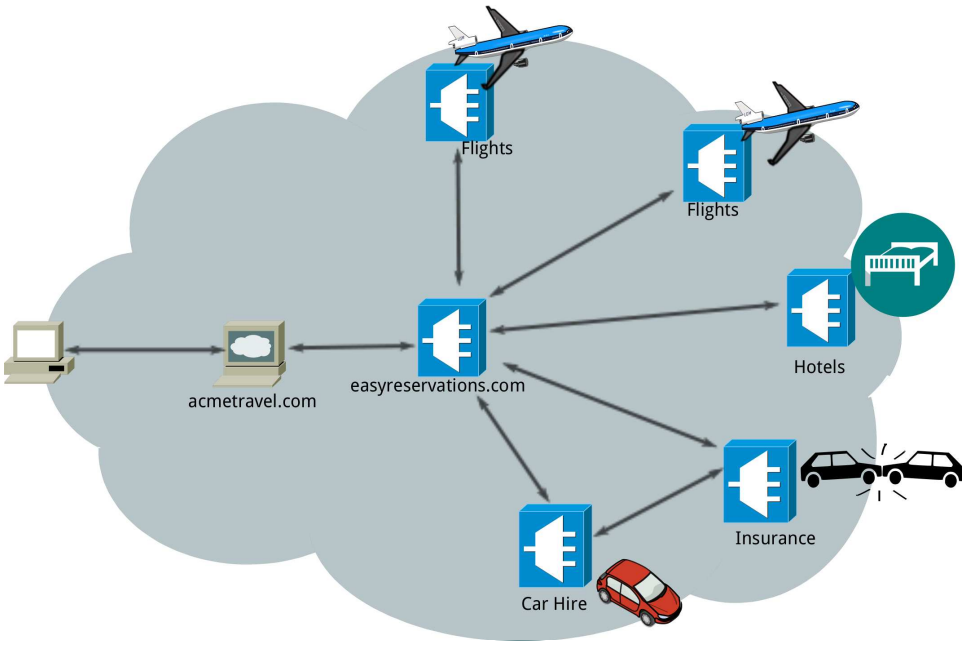

Figure 1.1: Example: Booking your next holiday on the Internet

With service providers moving towards multi-tenant architectures [Azeez et al., 2010; Bezemer and Zaidman, 2010; Tsai et al., 2010] and applications such as portrayed in the example being the norm, execution time predictability of web services demands an increased importance. Although Quality of Service (QoS) aspects of web services (such as scalability, availability, reliability and security) has been widely researched, few attempts have been made on achieving execution time QoS in web services middleware. Moreover, none of them would guarantee predictable execution times in a consistent manner.

1.1 The Problem

The execution of a service is managed by web services middleware (commonly referred to as SOAP engines) it is hosted in. The engine handles all aspects of request processing from listening for incoming requests, request processing, extracting parameters, invocation of service instances to the preparation of the response. Predictability of execution (which is the guarantee that a service invocation completes within a perceived deadline) is seldom considered a design goal in developing such web service engines. On the contrary, they are designed to achieve high levels of throughput (the number of service invocations handled within a given time) [Apache Software Foundation, 2009; Chapell, 2010; Sun Microsystems, 2009]. For instance, requests are accepted unconditionally and executed in a *best-effort* manner. Multiple requests are executed in parallel using processor sharing, using the *Thread-pool* concurrency pattern [Graham et al., 2004]. Figure 1.2 shows the average execution time of a service as resulted by worker threads in a thread pool, from a simple experiment we conducted.

The service we used has a linear execution time complexity ($O(n)$). We started off with 2 requests executing in parallel and increased the request count each time by five, until 47 requests were sharing the processor. The results obtained show that the average execution time increases proportionally with the number of requests executing in-parallel. This phenomenon is common in any kind of application including web services middleware, that conduct *best-effort* processing. This leads to longer and unpredictable waiting times that make web services unsuitable for applications with stringent execution time requirements. Furthermore, the development platforms and operating systems (OS) used to build and host such middleware, do not support execution level predictability. For instance, priority levels available to middleware may not orthogonally map onto OS level priorities. As a result, service execution maybe interrupted by other running processes or by house keeping activities within the development platform, such as garbage collection [Arnold et al., 2006].

Hosting web services in a cluster setup is a common way of improving response and execution times of services. Nevertheless, clusters do improve availability and reliability, but cannot guarantee predictability. Although distributing the request traffic among many hosts does improve conditions, the aforementioned issues still remain intact in the individual web service engines. Moreover, the dispatching algorithms used in them do not consider any predictability attributes such as execution deadlines or laxity, in the

decision making process that match requests to executors.

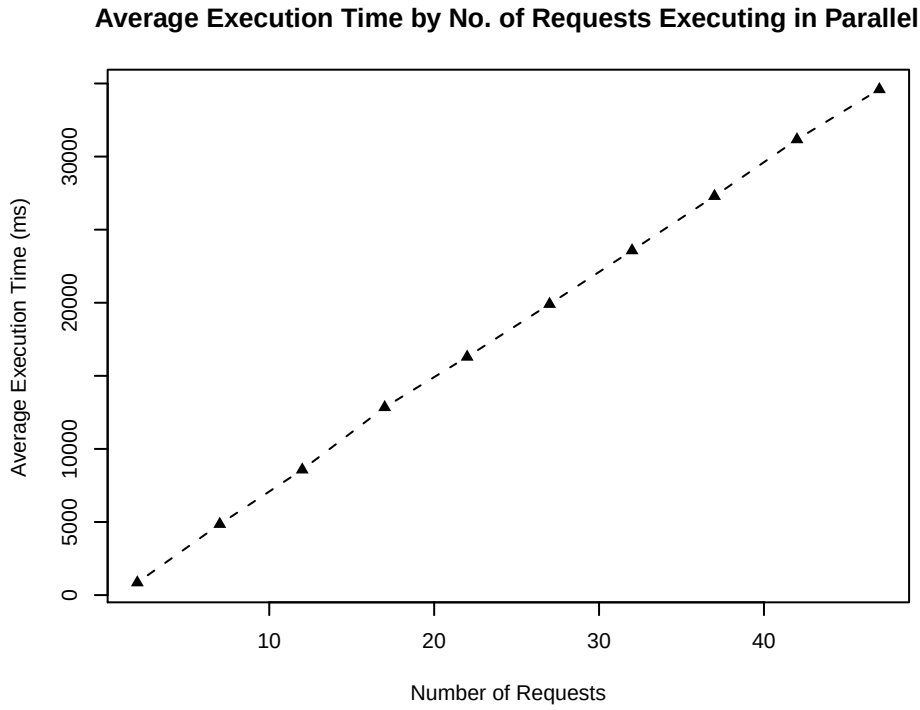

Figure 1.2: Average execution times by multiple requests processed

Due to these shortcomings, existing web services middleware [Apache Software Foundation, 2008, 2009; RedHat Inc., 2009; Sun Microsystems, 2009] is unable to guarantee execution time predictability. Consistent adherence to execution deadlines are of utmost importance for the use of web services for inter-application communication. Such shortcomings have hindered web services being adopted as a middleware for applications with stringent execution time requirements, such as disaster management, financial trading systems, industrial control systems, avionics, manufacturing execution systems and medical diagnostics systems. Such applications consider predictability of execution to be more important than throughput rates they could achieve. Although achieving certain levels of throughput is important, they trade-off throughput to achieve higher levels of execution time predictability. Therefore, such systems have been unable to benefit from the inherent features of web services, that has made it otherwise popular as a middleware platform.

Although there have been a few customised solutions that try to ensure execution level predictability [Helander and Sigurdsson, 2005; Mathes et al., 2009a,c], they are geared towards closed environments where task properties and arrivals are known at design time. They enable web services to be used with such applications with real-time properties, however the solutions are not applicable on the Internet, the open environment web services operate in, where request properties and arrivals are not know *apriori*. Moreover, the existing solutions claim to succeed in those specific closed environments, although important details such as the scheduling mechanism have been left out from the discussion, welcoming improvements to their methods and solutions.

1.2 Research Questions

The lack of support for execution time predictability is seen as detrimental to the success of web services being adopted as a middleware in applications with real-time requirements. Finding a solution to this problem needs to be tackled at multiple levels. As part of this process, we address the following research questions. With the first research question we introduce the notion of a deadline into the simplest form of web service middleware to explicitly select and schedule requests based on their deadlines. The second research question investigates the possibility of introducing this to a cluster setup and incorporate predictability based decision making into its request dispatching process. Implementing these techniques require software engineering techniques and tools that are generic and applicable to all types of web service middleware in use. Hence, the third question investigates the system building aspect of the solution. With the final research question we investigate the use deadline based scheduling for differentiation in systems where tasks may miss their deadlines and analytically model a deadline based scheduling system to obtain advanced performance metrics.

A) How can predictability of execution be achieved in stand-alone web services middleware?

The simplest web services deployments are stand-alone servers. Ensuring predictability on a single host would be the first step at achieving predictability in more complex configurations. Predictability requires execution of a request be completed within a perceived deadline. Such a feat could only be guaranteed by explicit scheduling of requests to achieve their respective deadlines. Nevertheless, with the open environment web services operate in, achieving processing deadlines of requests with no prior knowledge of

their properties, is a challenging proposition. Successfully accommodating a processing deadlines of unknown requests is a significant achievement as existing solutions that support such processing deadlines, are able to do so only with the knowledge of request properties and in closed systems.

B) How can predictability of execution be achieved in cluster based web service deployments

Web services deployments commonly use a cluster based setup to increase availability, reliability and for better response times. It is common for all cluster servers to have replicas of web services hosted. In such deployments, request dispatching takes place using traditional deployment algorithms that balance the load among the executors. Supporting predictability of execution requires the individual executors to ensure the requested deadlines can be met. This question, investigates the possibility of incorporating predictability attributes such as the execution deadline and laxity in the request dispatching decisions. The significance of the problem is incorporating the predictability attributes into the dispatching process and achieving the perceived level of predictability in the open environments web services are used in.

C) How can web services middleware products be engineered to have predictable execution times

With the first and second research questions, we ascertain the need for purposeful scheduling of requests and the requirement for selecting requests with guaranteed execution to meet their deadlines. Putting these concepts into practice requires web services middleware to be built for predictability. The challenges in building middleware to achieve predictability of execution or enhancing existing middleware will be different from engineering information systems. For instance, the change of goals from throughput to predictability requires a different school of thought and requires a more sequential processing oriented designs. Nevertheless, given the amount of requests to be processed, a certain amount of concurrent processing needs to be accommodated. The challenge here is to have a balance between these two aspects and build a system that achieves both the goals. Moreover, techniques and tools used in the development process of conventional applications will not be applicable in building such systems. With the third question, we investigate on such enhancements, software engineering techniques, best practices and tools to use in building web services middleware with predictable execution times.

D) How can performance models for systems using deadline based preemptive scheduling be derived and compared with other techniques

The performance of scheduling techniques are modelled and measured using performance attributes such as waiting time of a request. Although the main performance attribute associated with deadline based scheduling is deadline miss rate, the possibility of deriving other performance attributes such as waiting time is investigated with this question. Deriving such attributes require it to be modelled as a stochastic system. Although a few such performance models for deadline based scheduling systems can be found in literature, they consider the service times to be exponentially distributed and consider the system to be non-preemptive. Modelling preemptive execution is a challenging task on its own. When preemptions are made on deadline based decisions at runtime, the complexity is further increased. This question investigates the possibility of deriving a performance model for such a deadline based preemptive scheduling system.

1.3 Assumptions

For this research, our scope of achieving predictability is limited to request processing and execution that happens within web services middleware, due to the limited time frame. The execution of a web service may span across multiple application boundaries. For instance, part of its execution might have to do with fetching data from a database and as a result, a portion of the execution takes place within the database management system. We consider the predictability of such applications outside of the middleware as out of scope for this research. Indeed, achieving predictability on such applications is a research area on their own, based on their design and architecture. Within the scope of this research, we consider the execution time in such an external application as subsumed within the overall execution time of the web service invocation. Furthermore, we make the assumption that requests will experience any delays on the network. In this thesis our use of the terms request, task and job are synonymous and is used to refer to a web service request that is received by the middleware.

1.4 Limitations of Existing Solutions

This section highlights some of the limitations in existing solutions that ascertain the requirement of the research questions mentioned earlier. A more comprehensive discussion could be found in the respective related work sections within Chapters 3 to 6.

Execution level QoS in stand-alone web services middleware

Many of the existing work on execution time QoS in stand-alone web services make the assumption that the underlying middleware ensures the required QoS levels and act as service brokers [Ran, 2003; Tian et al., 2003] or facilitate service compositions [Zeng et al., 2003, 2004] based on QoS data. The few attempts that try to achieve differentiated request processing [Ching-Ming Tien, 2005; Sharma et al., 2003] does not consider an execution deadline as a QoS parameter. Some of the existing work employ admission control checks [Carlstrom and Rom, 2002; Dyachuk and Deters, 2007; Elnikety et al., 2004] to control the execution of requests. However, they do not consider predictability attributes such as execution deadlines and laxity of a request in the decision making process. In summary existing work achieve a level of differentiated request processing based on functional and non-functional attributes such as type of task (request processing versus security processing), nature of client (paying versus free) and type of devise used (mobile versus PC). Admission control mechanisms used do not However none of them supports an execution deadline or considers the laxity property of request in the differentiation criteria.

Execution level QoS in cluster based web services deployments

Out of the myriad of QoS attributes in web services, execution time is the most important attribute for predictability. Existing work on achieving better execution times on clusters have been mostly focused on improving response times by balancing or unbalancing the loads among cluster servers. Many of them use high level request classification schemes [Cardellini et al., 2003; Colajanni and Yu, 2002] to distribute requests evenly among cluster members or to give preference to one type of traffic over another. Similarly, others map requests to servers based on request properties such as the size of a request [Ciardo et al., 2001; Harchol-Balter et al., 1999]. Some use admission control

mechanisms and differentiated processing to achieve probability based measures of execution times specified in service agreements [García et al., 2009; Pacifici et al., 2005] and a few use heuristic techniques [Cao et al., 2010; Gmach et al., 2008] to achieve the same. Commonly, all work mentioned seem to achieve some differentiated processing in request execution, yet fails to guarantee predictable execution times in a repeatable and a consistent manner. Moreover, the decisions made in dispatching requests or in admission control, do not consider an execution deadline or considers the laxity of a task in the process.

Predictability of execution in existing web services middleware

To achieve execution level predictability in an open environment, three important steps must be ensured. Requests must be selected with a guarantee on meeting their deadlines, selected requests must be explicitly scheduled to meet the deadlines and finally the middleware must employ some method of differentiation in its request execution. Moreover, all of its activities must be supported by development platforms and operating systems that have real-time features. Existing specialised middleware solutions fail achieve one or more of these these steps in ensuring predictability. wsBus [Erradi and Maheshwari, 2005] contains an admission control mechanism and claims to use priorities to differentiate processing. However, there is no evidence of scheduling based on deadlines or how the priorities are enforced. Some of the more specialised middleware solutions [Helander and Sigurdsson, 2005; Mathes et al., 2009a,c] exhibit real-time features, in supporting requests with processing deadlines. However, they are custom made for closed environments and support periodic and aperiodic tasks with the assumption that task properties are known at design time of the system. The techniques they employ does not suit the open environment with unknown task properties.

Analytical models for deadline based scheduling systems

A few attempts at analytically modelling deadline based scheduling systems could be found in literature. Work of [Li et al., 2007] considers a non-preemptive system using earliest deadline first scheduling and proposes grouping of requests with similar deadlines together, to minimise the loss rate. Work of [Kargahi and Movaghar, 2006] uses the same scheduling principle and presents a performance model for deadlines at beginning and end of service for $M/M/m$ and $M/M/1$ type of queues, respec-

tively. [Lehoczky, 1996] attempts at incorporating the laxity property into a queueing model. The model presented considers a $M/M/1$ system and tries to minimise the loss rate by considering the laxity property when scheduling requests. A priority based multi-class scheduling model based on earliest deadline first scheduling for a non-preemptive $M/G/1$ system is presented in [Chen and Decreusefond, 1996]. They try to achieve differentiated waiting times for the different request classes based on their priorities. Many of the related work in this area consider the target system to be $M/M/1$ or having exponentially distributed service times. While this estimation may hold true when requests properties are known *apriori*, it misrepresents the unknown request properties and the open environment web services are typically used within. Moreover, the service time distribution will be based on the nature of tasks the service performs, which can be different in each case. The performance model presented in [Chen and Decreusefond, 1996] represents such an environment and considers the scheduling to be non-preemptive. This can be highlighted as a limitation, as deadline based scheduling could also be used with preemptive systems.

1.5 Research Contributions

In addressing the aforementioned research questions, we make the following contributions.

Predictability of execution in stand-alone web services middleware

We present an admission control mechanism and a deadline based scheduling method, to be used in stand-alone web services middleware. We provide an analytical model based on real-time scheduling principles that calculates the schedulability of a request given its deadline requirement and the requests already accepted for execution. The schedulability check considers the laxity property in accepting requests for execution and gives a guarantee on meeting their deadlines. Real-time scheduling principles are typically used in designing static schedules for closed systems. The uniqueness of the proposed solution lies in the fact that they are used in a highly dynamic and open environment, at run-time. A schedulability check algorithm based on the analytical model selects request based on their laxity and only selects request with execution deadlines that can be met given the current conditions. The selected requests are scheduled for

execution using Earliest Deadline First (EDF) scheduling algorithm. Such purposeful selection and scheduling of requests ensures that execution deadlines of requests can be achieved by the system. With these techniques used for achieving predictability, there is an unavoidable reduction of throughput. However, we ensure the resultant throughput levels to be within an acceptable range, when the nature of application is considered.

Predictability of execution in web service clusters

We present four request dispatching algorithms to be used in cluster base web service deployments that consider the processing deadline as a parameter in dispatching decisions. Two of the algorithms dispatch requests in a content-blind manner and the schedulability of a request with a chosen executor is considered prior to dispatch. The remaining two algorithms carry out content-aware dispatching. One uses the task size to match a request to an executor and the other distributes requests among executors to increase the range of laxities at each executor. All four algorithms make use of the laxity property when the dispatching decisions are made. The laxity based request selection of the four algorithms ensure that the requests at each cluster server comprise of a wide range of laxities. This enables more requests with overlapping deadlines to be scheduled together by delaying or phasing out the requests of requests with larger laxities. Selected requests are executed at each server using EDF scheduling algorithm. The four algorithms represent a majority of the widely used dispatching techniques that either balance or unbalance the load of a cluster and therefore are a good example of how predictability attributes such as a laxity could be considered as part of the dispatching decisions.

Building web services middleware with predictable execution

The software engineering aspect of the overall solution is also considered as a main contribution of this thesis. A real-time scheduler component makes use of a priority model and custom made real-time thread pools to achieve fine-grain control over the execution of requests in the servers. Moreover, algorithms and system designs to implement the schedulability check and the deadline based scheduling algorithm at the middleware level is also presented as contributions. As, guaranteeing execution level predictability requires a more serialised approach to request execution, separate lanes of execution has been introduced into the web services middleware. Real-time worker

threads with elevated priority levels executing in these lanes gives better control of their execution to the real-time scheduler. In memory logging and debugging techniques that minimises Input/Output activities are used to ensure priority inversions are kept to a minimum. Such software engineering techniques, design patterns, algorithms and tools for achieving predictability is considered to be the main contribution.

Advanced performance modelling of earliest deadline first scheduling in web services

We present an analytical model based on queueing theory to measure the performance of a priority based preemptive $M/G/1$ system using earliest deadline first scheduling technique. The model is an extension to the work of [Chen and Decreusefond, 1996], where the referenced model is extended to a preemptive deadline based scheduling system. The model approximates the mean waiting time of a request belonging to a particular priority class where the mean waiting time is based on the execution of higher and lower priority requests already at the system, higher priority requests arriving at the system subsequently and having prior service and the mean residual service time experienced by the priority class. The preemptions are approximated as part of the mean completion time experienced by a request of the considered priority class and it is encapsulated in the definition presented for the mean residual service time experienced by the request. To our knowledge this is the first attempt at approximating the performance of such a system using queueing theory. The significance of this model is that it is independent of the service discipline and therefore is applicable to any scheduling technique. Moreover, the model and its approximations are valid not only for web services, but any other system that uses a similar queue and earliest deadline first scheduling.

1.6 Thesis Structure

The remainder of this thesis is structured as follows

- Chapter 2 presents a background on core concepts used in this thesis. This includes an introduction to web services, service oriented architecture and the architecture of web services middleware. It is followed by an introduction to real-time tasks, real-time scheduling principles and a discussion on deadline based

scheduling in detail, parts of which are used by the analytical models presented in this thesis.

- In Chapter 3 we present an admission control mechanism and a deadline based scheduling method to be used in stand-alone web services middleware. We use real-time scheduling principles typically used at design time, to achieve predictability in real-time systems. They are used for admission control and request scheduling, in an environment where task properties and their arrivals are unknown. These techniques introduced into a middleware product and evaluated with different traffic conditions by comparing with its unmodified version. Empirical results show that more than 96% of the request deadlines could be met while accepting more than 20% of the requests in very high traffic conditions.

- Chapter 4 extends the admission control mechanism and the deadline based scheduling method into a cluster environment and incorporate the laxity property of a request in the dispatching process. We introduce four different request dispatching algorithms that considers laxity, and dispatches requests only if their deadlines could be met. We implement these in two middleware products and compare them to popular techniques such as round-robin and class-based dispatching, evaluating them under different traffic conditions. Empirical results obtained confirm that the proposed methods achieve 95% of the deadlines compared to less than 10% of the deadlines by others.

- Chapter 5 discusses the engineering aspect of introducing predictability of execution in to actual middleware products. We present generic software engineering techniques, algorithms and tools that can be used for this purpose. Moreover, we provide a set of guidelines that can be followed in identifying and making the changes in existing web services middleware products.

- Chapter 6 provides details of an analytical model for a preemptive $M/G/1$ system which uses earliest deadline first scheduling principle. The system supports arbitrary number of priority levels and the priority governs the waiting time and loss rate experienced by requests in each class. Compared to similar systems with non-preemptive scheduling, the proposed model achieves better waiting times and loss rates. Analytical and empirical results show that deadline based scheduling in such a configuration achieves a better balance in terms of higher priority requests

experiencing less waiting time and loss rates, without lower priority requests being led into over-starvation.

- Chapter 7 summarises the proposed approach for each research question and discuss potential directions for future work based on the solutions presented, before the conclusion.

Chapter 2

Background

This chapter provides the background information of technologies and techniques that the proposed research is based on. A knowledge of these are a necessity to understand the problem, questions addressed and the solutions presented as part of this research.

2.1 Web Services

Web Services are self-contained software components that are accessible over a network and perform designated tasks for a user or an application [Papazoglou, 2008]. Their features such as, being self-describing [Gurugé, 2004], loosely coupled [Weerawarana et al., 2005], transport agnostic [Cerami and St Laurent, 2002], discoverable [Chappell and Jewell, 2002] and platform independent [Marks and Werrell, 2003] have made them the most popular choice for communication in distributed systems.

Web Services use Web Service Description Language (WSDL) [Christensen et al., 2001; Weerawarana et al., 2002] which is an XML based specification to describe themselves. It contains information about the service, operations available, parameters that are accepted by each operation, data types of the parameters, return values, messaging schemes and protocols used. Although being human readable due to its XML nature, the WSDL description is intended to be accessed programmatically by applications [Gurugé, 2004]. Web services are considered to be loosely coupled as their definition or data exchange do not rely on any underlying platform, operating system or the program language they are implemented in. This enables web services to be highly

inter-operable [Weerawarana et al., 2005].

Earlier versions of web services only supported standardised XML-based documents such as SOAP and XML based Remote Procedure Call (RPC) format, for data exchange [Chappell and Jewell, 2002]. Over the years, the widened use have seen relatively newer and lighter weight technologies such as Representative State Transfer (REST) [Fielding, 2000; Richardson and Ruby, 2007] based services, description frameworks such as JavaScript Object Notation (JSON) and binary based object representations such as Message Transmission Optimisation Mechanism (MTOM) [Gudgin et al., 2005; Jayasinghe and Azeez, 2011] being widely supported for data exchange. Similarly, SOAP-based message exchange was facilitated by HTTP, which enabled the data packets to easily get through firewalls and routers over the Internet. Since then, a myriad of protocols such as Simple Mail Transport Protocol (SMTP), File Transfer Protocol (FTP) and Java Message Service (JMS) support web services data exchange [Vogels, 2003].

Web Services are registered and discoverable through its own XML-based directory service, named Universal Description Discovery and Integration (UDDI). Service providers can register their services in the UDDI registry and clients could look-up services using different criteria both static and dynamic [Graham et al., 2004]. Static attributes queried on are typically information such as network location and protocol, where the directory will return the same web service information everytime. Dynamic attributes such as response time, security specifications, price could also be used to query for services and the registry would return services based on what is offered at the given time.

As its specification does not rely on any development platform, operating system or hardware specification, web services are considered platform independent. As XML implementations can be found on most platforms, web services can be hosted and accessed on most operating systems and devices used. Moreover, the text based nature of XML, enables web services to be easily introduced on to newer platforms and devices.

A web service in its functionality can take many forms. It could be implemented to perform a specific task or it could represent a complete business process. Similarly, a service could be an external representation to a resource such as a device or an application. Whatever form it may take in terms of functionality, its architecture remains the same [Papazoglou, 2008].

2.1.1 Service Oriented Architecture

Service Oriented Architecture (SOA) is a design concept that enables software to expose part of their functionality as services to other applications or services [Erl, 2007]. Although this is a general concept that could be applied to any type of technology, web services are the best example for following the message oriented delivery model introduced by SOA.

Figure 2.1 illustrates the different roles web services play in SOA, their relationships and their operations. A service provider would publish descriptions of the services they provide to a web services registry. A client would search for web services on the registry using different parameters such as functionality and QoS attributes. The service registry would return descriptions of services to the client, that matches the search criteria specified. Finally, the client would use the information obtained from the registry to invoke the desired web service at the service provider.

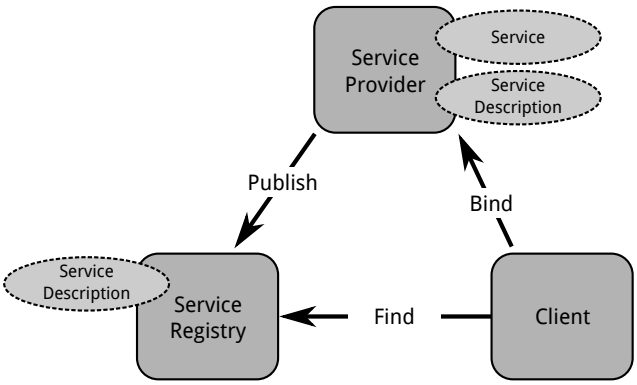

Figure 2.1: Roles of Web Services in SOA (based on [Papazoglou, 2008])

Organisations started embracing web services technology easily due to the short learning curve and the opportunities the technology presented them with, in terms of integration and collaboration between applications and systems that were existing simply as information silos. Moreover, its relatively minimal demand for infrastructure did not incur a significant additional cost in using the technology. Web services also gives its users flexibility, in terms of how they are implemented or deployed in an organisation. Its non-monolithic implementation contains a collection of technologies and options that could be picked depending on a user requirements.

Figure 2.2 illustrates the web services technology stack and how it fits in with applications in terms of being a middleware. The Network layer at the bottom represents the transport layer of the networking protocol stack. Transport protocols such as TCP are used by the transfer protocols web services use for data exchange. The application layer on top represents an application that makes use of web services as a middleware. The labels on to the right depicts the role of each layer within the stack.

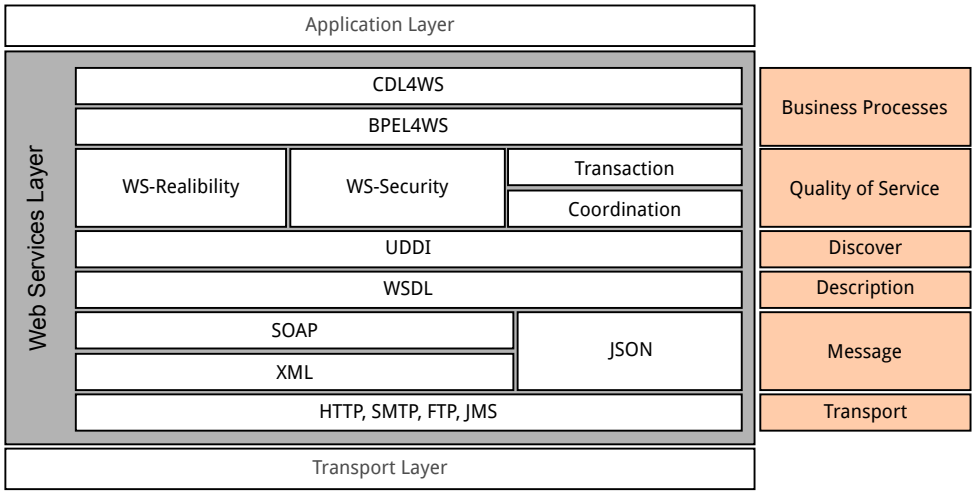

Figure 2.2: Web Services Layers

As illustrated, the web services protocol stack is a collection of related technologies, each playing a unique role in its operation. The bottom most layer in the web services stack is the protocol that transfer data between two web services middleware systems. There are numerous protocols that can be used here with HTTP being the most popular. A few technology choices such as XML, SOAP and JSON are available as data exchange formats for messaging purposes.

The information about a service, its operations, endpoints, ports, data types of parameters and their order are described using WSDL which uses an XML based notation. WSDL descriptions of a service can be found at the service provider and at a service registry. Service discovery is facilitated by UDDI, a public directory that service providers could use to advertise their services on. As mentioned earlier, clients query the registry with various parameters and the registry locates the service based on the criteria specified.

SOAP, WSDL and UDDI are considered as the core standards in web services [Papazoglou, 2008]. However, over the years many other features have been introduced to web services through other value added standards. Aspects such as Transactions, Coordination, Security and Reliable Messaging have been developed and accepted as standards for applications to use, although being optional for web service operations. Similarly, BPEL4WS (Business Process Execution Language for Web Services) [Khalaf and Nagy, 2002; Peltz, 2003; Weerawarana and Curbera, 2002] is an XML based language that facilitates the description of business processes as composite web services where the flow of execution can be defined in terms of conditional, sequential and parallel executions with supported exception handling. Complex business processes that many span across multiple organisational boundaries can be described and and handled using BPEL4WS and supporting middleware. Moreover, the description of collaborations that exists between different systems and organisations is facilitated by CDL4WS (Choreography Description Language for Web Services) [Kavantzas et al., 2005] which is also an XML based notation. CDL4WS describes the information exchange between composite services in a business collaboration, while BPEL4WS defines the composite services.

2.1.2 Simple Object Access Protocol

SOAP was invented as a solution to the problem of proprietary systems and protocols being used on heterogeneous infrastructure. Middleware used in distributed systems, such as CORBA [Vinoski, 1993], DCOM [Microsoft Corporation, 2012] and Java RMI [Oracle, 2010] mandate the use of binary based proprietary wire protocols, require proper runtime environments installed, properly configured and administered apart from the applications they facilitate in data exchange. SOAP was designed to have interoperability and it achieves this due to the text based format that it inherited from XML. Compared to protocols like Internet Inter-ORB Protocol (IIOP) which is the wire protocol of CORBA, SOAP is much more light weight, having only two fundamental properties in its operation. They included sending and receiving transport data packets (using HTTP or other transfer protocols) and processing XML messages that are used for data exchange [Scribner et al., 2000].

Since XML was widely adopted and supported by many platforms, SOAP had only minimal requirements for setup and operation. Although mostly associated with HTTP, SOAP can be used with many transfer protocols giving the flexibility for users. Its abil-

ity to make use of many of the protocols used on the Internet, makes it much easier for the messages to get through existing firewall configurations, requiring no additional setup time. The SOAP specification [Gudgin et al., 2007a,b; Mitra and Lafon, 2007] simply contains the basic structure of the message and the encoding mechanisms used. It does not mandate any specific semantics for implementation, thereby giving the freedom for its use in many types of applications ranging from messaging systems to RPC.

In its simplest form, SOAP is a stateless one way message exchange paradigm. However, applications have the flexibility on using it to create more complex interactions such as request/response and request/multi-response by combining multiple one way message exchanges facilitated by an underlying protocol or specified by the application that uses it. Figure 2.3 contains a sample listing of a SOAP message in the format found on the wire.

```
POST /axis2/services/FactorPrimesService HTTP/1.1
Content-Type: text/xml; charset=UTF-8
SOAPAction: "urn:primeCount"
User-Agent: Axis2
Host: localhost:8080
Transfer-Encoding: chunked


<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Header>
  <ns1:RealTimeParams xmlns:ns1="http://endpoint.testservice">
   <ns1:Deadline>70</ns1:Deadline>
   <ns1:Period>0</ns1:Period>
   <ns1:clientid>Client1</ns1:clientid>
   <ns1:ExecTime>28</ns1:ExecTime>
  </ns1:RealTimeParams>
 </soapenv:Header>
 <soapenv:Body>
  <ns1:primeCount xmlns:ns1="http://endpoint.testservice">
   <ns1:primeLimt>102155</ns1:primeLimt>
  </ns1:primeCount>
 </soapenv:Body>
</soapenv:Envelope>
```

Figure 2.3: Sample SOAP Message Listing

2.1.3 SOAP Message Structure

Figure 2.4 shows the structure of a SOAP message with a sample listing. The basic structure of a SOAP message constitutes of a header portion which is considered optional and a body section that carries the payload or the message that is intended for the remote application.

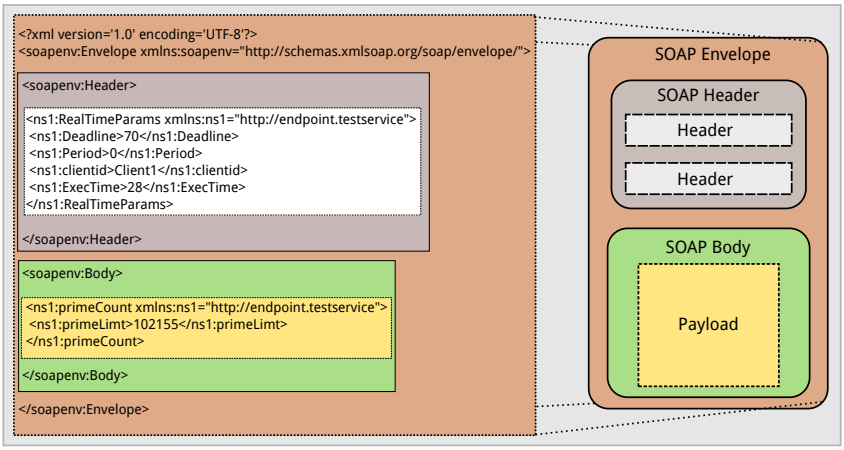

Figure 2.4: SOAP Message Structure

The header element separates the metadata from the actual payload and may contain data items from the optional SOAP extensions such as Web Service Addressing [Box et al., 2004], Realiable Messaging [Davis et al., 2006] and Web Service Security [OASIS, 2006]. However, the use of the header elements are not limited to the standardised SOAP extensions, rather they could be used by programmers to transfer any metadata independently to the payload contained in the SOAP body. The payload can be purely XML based textual data or have elements with binary content such as when when SOAP with attachments (using MTOM) is used for transferring data in the form of images and other document types.

2.1.4 Web Services Engine

Web Services are deployed in a server supported by a container application. Known as a Web Services Engine or SOAP Engine, this container application facilitates the processing and invocation of service instances. It has a broader role to play on the server side than the client side. The architecture of a SOAP engine can be based on various requirements. Some are optimized for performance and some have been designed with extensibility in mind. Although there are such subtle differences in their design, it is possible to identify parts of discrete functionality to do with the fundamental task of message processing, as illustrated in Figure 2.5.

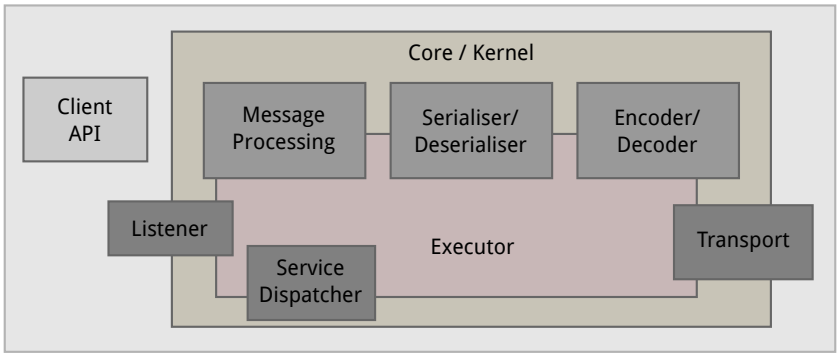

Figure 2.5: Architecture of a Generic SOAP Engine

At its core, the Web Service Engine has an executor component that handles the execution of a request through the modules. Typically an executor consists of a pool of worker threads that handle the execution of requests. Once a task is assigned to a thread, the execution through each module is managed by the same worker thread. This happens by default, in a *best-effort* manner in every SOAP engine.

SOAP engines can be used both at the client and server ends. Although the processing that takes place inside an engine is built around SOAP messages, it is a common practice to use an internal object or a data structure to represent and/or encapsulate SOAP messages inside the engine. These will contain some additional information used by the other modules within the SOAP engine.

Listener

The listener module is the gateway to the Web Services Engine. It continuously listens for incoming requests and hands them over to a Transport module. Depending on the network transport protocols supported by the web service engine, there maybe multiple listeners (i.e HTTP, SMTP, JMS, etc.) active simultaneously on different ports of a server.

Transport

Upon receiving a request, the transport module parses the packets and creates an internal data structure that represents and/or encapsulates the SOAP message. Some of the typical data that gets stored in the internal representation would be the transport protocol

information and the SOAP action. The internal structure containing the message is passed onto the other modules in the SOAP engine. When a message is sent out from the SOAP engine. The completed SOAP message is handed over to the module, contained in the internal representation. The module processes it and prepares the data packets to be sent to the network.

Message Processing Module

The Message Processing Module parses and carries out processing based on the headers in the SOAP messages. The SOAP headers are used to exchange various meta-data and can be used for various purposes such as enforcing authentication mechanisms and achieving reliability. Several WS-* standards make use of the headers. Conformity to such standards and related processing happens within this module.

Serialiser / Deserialiser

Serialisation refers to the process of transforming a SOAP message into a byte form that could be transported over the network using a transport protocol. This process takes place when either a client makes a request or a server sends a response to a service invocation. When a server receives a request, the de-serialisation process takes place. The data received from the network is extracted by the transport modules and is handed over. The SOAP message is reconstructed from the data and passed onto the subsequent modules in the SOAP engine.

Encoder / Decoder

Encoding refers to the process of transforming programming language specific data types and values to their ‘mapped’ XML representation. This process takes place when a client makes a web service call and the parameters are marshalled. Furthermore, this process also takes place when a server wants to send a result of a web service call back to the client. Decoding refers to the reverse process of transforming XML representations into data types and values of a particular programming language. This takes place on a client, when a reply is received for a web service invocation or at a server when a client request reaches the SOAP engine.

Dispatcher

At the server end, the same SOAP engine is typically used to host more than one web service, each identified by a different endpoint reference. When a request is received at the server, the dispatcher module is responsible for deciding which of the deployed web services is the recipient of the message. Once the service is located, the dispatcher uses the de-serialised and decoded content of the message to invoke the located service.

Executor

A SOAP engine is capable of handling multiple requests for SOAP message processing. It is common for a server to have multiple web services deployed. Therefore, receiving multiple requests for multiple services is commonality. SOAP engines use an executor module with a pool of worker threads to internally handle each of the requests separately. Once the request is received, a worker thread is assigned to it and it is responsible for coordinating the functionality across each of the modules until the task is completed. A reply containing the result of the service invocation or details of an error encountered being sent back to the client and housekeeping activities such as closing the connections, signifies the completion of request processed. After completing the request, the thread is either destroyed or included back into the pool to take on another request, based on the SOAP engines design.

Client API

Most SOAP engines provide client side functionality through a set of Application Programming Interfaces (API). These would include additional functionality such as the generation of local stubs using WSDL definitions available on a server. Client APIs make use of serialisation and encoding modules to prepare the web service request at the client end. Similarly, the reverse process happens when a reply is received from the server.

2.2 Real-time Systems

Real-time systems consider the predictability of execution equally important as the correctness of an operation. Such systems mandate the completion of a task within a per-

ceived deadline, where even a correct result obtained with a deadline miss is considered useless [Buttazzo, 1997; Stankovic et al., 1998]. This area of computer science contains scheduling principles and techniques that facilitate in achieving such deadlines, consistently. In that light, we look at some of the principles and techniques used in real-time systems prior to the discussion of our solution.

2.2.1 Real-time Tasks

A real-time task is the smallest processing entity in a real-time system. They are characterised by certain timing properties [Buttazzo, 1997].

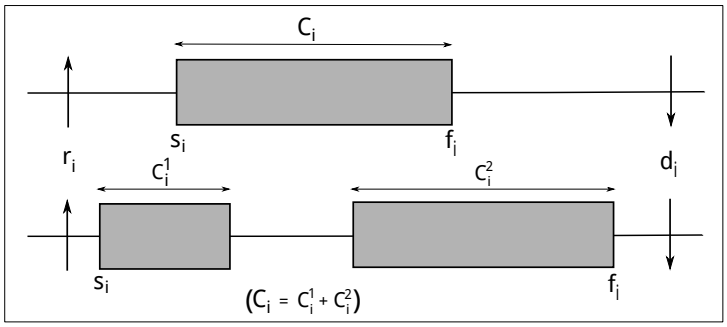

Figure 2.6: Timing Properties of a Real-time Task

Figure 2.6 shows two real-time tasks with main timing properties highlighted. Depending on the type of scheduling used, a task may execute in a preemptive or a non-preemptive manner. The first task in the Figure 2.6 has a non-preemptive continuous execution, whereas the second task has executed with one preemption.

- **Arrival time** (r_i): The time a task appears at the system. At this time a task is available for execution. Arrival time is also referred to as *release time*.
- **Deadline** (d_i): The ultimate time limit to complete the execution of a task. Depending on the type of task, completing the execution beyond this limit may render the task useless to the system.
- **Start Time** (s_i): The time at which a task starts execution in the system.
- **Finishing Time** (f_i): The time at which a task finishes execution in the system. This signifies the completion of the work with our without preemption.

- **Computation time** (C_i): The total time required to complete the execution of a task. If preemptive execution is allowed, computation time does not include the time a task spent being preempted.
- **Laxity** (X_i): Also referred to as *slack time*, Laxity is the maximum time the execution of a task could be delayed without missing its deadline. Laxity can be calculated either as time ($X_i = d_i - r_i - C_i$) or as an indicator given by the ratio between the deadline and computation time ($\frac{d_i}{C_i}$).

Types of Real-time Tasks

Real-time tasks could be classified according to the nature of their deadline [Arora, 1997; Buttazzo, 1997].

- *Hard Real Time* tasks- Deadlines cannot be missed. Missing it will make the result unusable and could lead to fatal errors.
- *Soft Real Time* tasks - Missing a deadline will not result the task being unusable, however there may be a penalty involved with it.
- *Firm Real Time* tasks - The value of the outcome of the task diminishes over time. The sooner the tasks finish their computations, the higher the reward is.

Tasks could be further classified according to their frequency of occurrence [Arora, 1997; Mohammadi A. and Akl S. G., 2005; Stankovic et al., 1998].

- *Periodic* - Tasks that are released at regular intervals, based on a fixed rate. An example would be periodically reading a value off a sensor.
- *Sporadic* - Tasks that are released at random intervals but with a known bounded rate. The bounded rate is characterized by a minimum inter-arrival period.
- *Aperiodic* - Tasks that are released at random intervals and with an unbounded rate. An example would be a task that occurs due to human interaction with the system.

Most real-time systems can be found in closed environments such as in embedded systems [Buttazzo, 1997]. Due to the closed nature, tasks in such systems and their properties are known to system designers prior to the system being built. Having such information at design time, enables them to plan for resource requirements of tasks, execution precedence and design static schedules [Stankovic and Rajkumar, 2004; Stankovic et al., 1998].

2.2.2 Real-time scheduling

There are several scheduling policies available in real-time scheduling and each of them are better suited for different task types. We present two such widely used policies that are proven optimal for certain scenarios [Liu and Layland, 1973].

Rate Monotonic

Rate Monotonic (RM) is an optimal fixed priority scheduling policy where priorities are assigned to tasks based on the frequency of occurrence. Therefore, this scheduling policy is ideal for recurring tasks where the frequency is known *apriori*. Herein, priorities are assigned (fixed) and order of execution is decided prior to the start of execution. RM is considered as an optimal static algorithm, in a sense that no other fixed priority algorithm can schedule a task set that cannot be scheduled by it. However, its schedulable bound is less than 100%. Schedulable bound is the maximum Central Processing Unit (CPU) utilisation level achieved by a set of tasks, up to which deadlines of all tasks is guaranteed to be met. While the policy works well with periodic tasks, it could be disrupted by aperiodic and sporadic tasks in the system.

For a given Task τ_i, with a worst case execution time is C_i, a period of P_i, the fraction of CPU time spent in processing τ_i is C_i/P_i. The total CPU time spent in executing *n* tasks is:

$$U = \sum_{i=1}^{n} C_i/P_i$$

According to [Liu and Layland, 1973] the worst case schedulable bound W_i for *n* tasks is:

$$W_n = n(2^{1/n} - 1)$$

If $\sum_{i=1}^{n} C_i/P_i \leq n(2^{1/n} - 1)$, where *n* is the number of tasks to be scheduled, then the RM policy will schedule all the tasks to meet their respective deadlines. RM policy has a complexity of O((N + α)^2) in the worst case, where *N* is the total number of requests in the hyper period of *n* periodic tasks in the system and α is the number of aperiodic tasks in the system [Mohammadi A. and Akl S. G., 2005].

Earliest Deadline First

The EDF scheduling policy considers deadlines of tasks in assigning priorities. The priorities of tasks are not decided at design time (as in the case of RM) and can change dynamically at runtime. Priorities are typically assigned to tasks on their arrival at the system. However the arrival of a task could result in a change of priorities in the tasks already in the system. The deadline based priority system means that a higher priority task with an earlier deadline could preempt a lower priority task. As a result, the order of execution may change at any time. This process results in a higher scheduling overhead in EDF compared to a static policy such as RM. However, these characteristics also make EDF a better choice for systems with aperiodic and sporadic tasks. Moreover, such features enable EDF to achieve a schedulable bound of 100% for all tasks. EDF is considered as an optimal dynamic algorithm, in a sense that no other dynamic priority algorithm can schedule a task set that cannot be scheduled by it. However, one drawback of EDF is that there is no guarantee of which task would fail in overload conditions, whereas with RM lower priority tasks will always fail in overload conditions.

For a given Task τ_i, with a worst case execution time of C_i and a period of T_i, If

$$\sum_{i=1}^{n} (C_i/T_i) \leq 1 \tag{2.2.1}$$

where *n* is the total number of tasks, it is feasible to schedule the set of tasks to successfully meet their deadlines. In other words, if the total CPU utilisation of the task set is less than or equal to 100%, it is deemed feasible to use EDF to schedule the set of tasks. EDF has a complexity of O((N + α)^2) in the worst case, where *N* is the total number of requests in the hyper period of *n* periodic tasks in the system and α is the number of aperiodic tasks in the system [Mohammadi A. and Akl S. G., 2005].

Recall that our attempt is to use real-time scheduling principles in web services middle-

ware to achieve predictability of service execution. Herein, the greater challenge is to use scheduling policies typically used in closed systems where most task properties are known *apriori*, in a highly dynamic environment where task properties are not known. The unknown nature of web service requests means that tasks will be aperiodic and the best scheduling policy for such tasks will thus be EDF. Therefore, next we dwell further into important concepts behind EDF scheduling.

On a *uni-processor* system, assuming a *non-idling* and a *non-preemptive* system, if all tasks are ready at time *t=0*,

Theorem 1 (Jackson's Rule [Jackson, 1955]) *Any sequence is optimal that puts the jobs in order of non-decreasing deadlines.*

For a *uni-processor* system, having tasks with *arbitrary* release times, deadlines, execution times or *unknown* execution times,

Theorem 2 ([Dertouzos, 1974]) *The EDF algorithm is optimal in that if there exist any algorithm that can build a valid (feasible) schedule on a single processor, then EDF algorithm also builds a valid (feasible) schedule.*

Any valid schedule (a set of tasks that could be successfully scheduled within a given time period) can be converted into a valid EDF schedule by using a *'time slice swapping'* technique, where the order of tasks are interchange to arrive at an EDF schedule. Figure 2.7 illustrates this with a simple example. The example contains 3 tasks executed within a 20 time unit interval. This is assumed to be a synchronous schedule (i.e. all tasks share the same start time of *t=0*). Task *T1* has a deadline of 20 time units and an execution time of 8 time units, *T2* has a deadline of 18 time units and an execution time of 3 time units and *T3* a deadline of 17 time units and an execution time of 5 time units. In the given schedule, the task with the earliest deadline (*T3*), gets the CPU last.

When converting this to an EDF based schedule, the time slice of task *T1*, is swapped with the time slices of tasks *T2* and *T3*. As *T3* has the earliest deadline, it is executed first, followed by *T2* and then by *T1*. The schedule with 3 tasks is transformed into a valid EDF based schedule, meeting the deadlines of all 3 tasks.

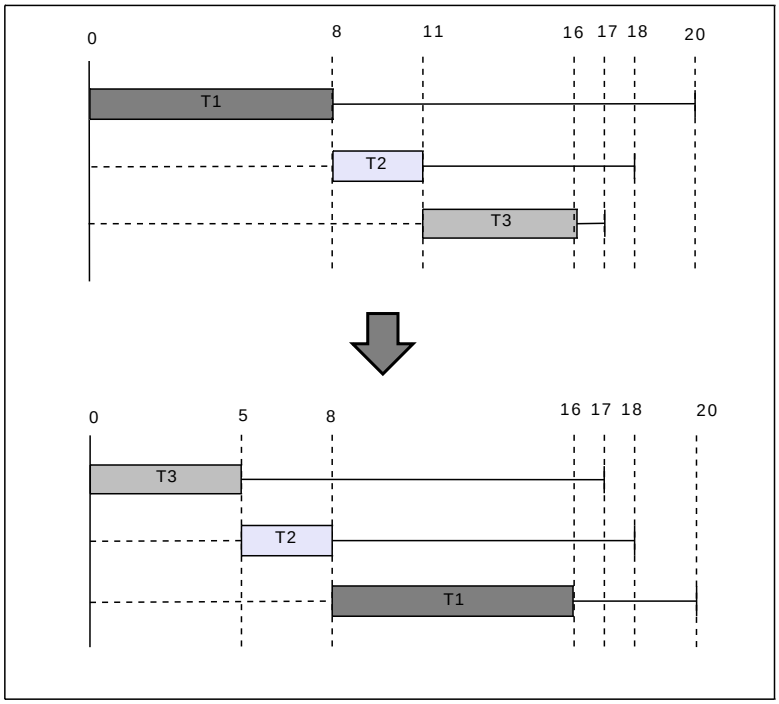

Figure 2.7: Transformation of a valid schedule to an EDF based valid schedule

The optimality of a real-time deadline based scheduling algorithm such as EDF is ensured by schedulability analysis (depicted in equation 2.2.1), a step carried out offline. Taking execution time requirements of tasks into consideration, this aids in analysing the feasibility of a schedule. As it is meant to be worked offline, it renders itself unsuitable in a web services scenario. However, certain principles it is based on can be used in devising a suitable online feasibility analysis (which is discussed in Chapter 3).

Two concepts considered in schedulability analysis, namely *processor demand* and *loading factor* [Stankovic et al., 1998] are defined here. Henceforth, we use a given task T_i, with release time of r_i, a deadline of d_i and an execution time requirement of C_i. Our proposed model for schedulability analysis (presented in Chapter 3) is based on the following definitions.

Definition 1 For a given set of real-time tasks and a semi-closed interval of time $[t_1, t_2)$, the processor demand (h) for the set of tasks in the interval $[t_1, t_2)$ is

$$h_{[t_1,t_2)} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k. \tag{2.2.2}$$

Definition 2 For a given set of real-time tasks the fraction of the semi-closed interval $[t_1, t_2)$ needed to execute its tasks is considered as its *loading factor* (u) that is,

$$u_{[t_1,t_2)} = \frac{h_{[t_1,t_2)}}{t_2 - t_1}. \tag{2.2.3}$$

Definition 3 The loading factor of the maximum of all such intervals, is considered as *absolute loading factor*, that is,

$$u = \sup_{0 \le t_1 \le t_2} u_{[t_1,t_2)}. \tag{2.2.4}$$

As an example, applying these to the tasks in the EDF schedule obtained in Figure 2.7,

$u_{[0,7)}$	=	$\frac{5}{7}$
$u_{[0,10)}$	=	$\frac{5+3}{10} = \frac{8}{10}$
$u_{[5,16)}$	=	$\frac{3+8}{16} = \frac{11}{11}$
$u_{[0,16)}$	=	$\frac{5+3+8}{16} = \frac{16}{16}$
$u_{[0,20)}$	=	$\frac{5+3+8}{16} = \frac{16}{20}$
u	=	$\frac{16}{16} = 1$

Table 2.1: Loading factor computation for the job set of EDF schedule in Figure 2.7

Theorem 3 ([Spuri, 1995]) *Any set of real-time tasks is feasibly scheduled by EDF algorithm only if*

$$u \le 1. \tag{2.2.5}$$

2.3 Summary

This chapter provided a background on Web Services and Web Services Middleware which our research is mainly based on. We achieve predictability of execution in web services by introducing real-time scheduling techniques into web services middleware. Therefore, a detailed discussion on the basics of real-time scheduling principles was also included in the chapter. In the next chapter, we investigate the first research question of achieving predictability in stand-alone web service middleware.

Chapter 3

Predictability of Execution in Stand-Alone Web Services Middleware*

Web services are witnessing a tremendous growth in their usage on the Internet, with many things being offered as services. Web services middleware contain many optimisations that enable them to achieve high rates of throughput with the increased demand for services. For instance, they unconditionally accept any request sent to them and make no differentiation in their execution. Requests are executed in a *best-effort* manner, using processor sharing. However, such techniques adversely affect the predictability of service execution, with service invocations returning highly unpredictable execution times. In this chapter we present an analytical model and a deadline-based scheduling technique that enable stand-alone web services middleware to guarantee predictable service execution times. We introduce the notion of an execution deadline which the middleware must adhere to in executing requests. Depending on the rate of request arrival, it is unlikely that deadlines of every incoming request can be catered for, given the unknown properties of requests in an open environment such as the Internet. The proposed analytical model based on real-time scheduling principles calculates the demand for processing resources, given the already accepted requests for execution. An admission control algorithm based on the proposed model selects requests for execu-

* Preliminary versions of the work presented in this chapter have been previously published in [Gamini Abhaya et al., 2009, 2010b, 2012].

tion based on their laxity property, considering the processor demand calculated. The proposed algorithm accepts a request for execution if their execution deadline can be met without compromising the deadlines of already accepted requests. Thereafter, selected requests are executed using the earliest deadline first scheduling principle. The predictability gain achieved by the proposed techniques are evaluated by implementing them in Apache Axis2 and compared with its unmodified version. Empirical results confirm that the enhancements allow the middleware to consistently achieve more than 96% of execution deadlines while withstanding high request arrival rates and accepting more than 18% of the requests for execution in the worst traffic conditions.

3.1 Motivation

Web services play an important role in the current distributed computing landscape. They are transforming the Internet to a fully automated web of autonomous applications that discover and communicate with each other without any user invention. However, this increased usage of web services is also witnessing new deployment models such as multi-tenancy [Azeez et al., 2010], where a single web services middleware is used to cater many tenants and their requests.

With the use of composite services, operations on a single application may result in hundreds of web services invocations to many other systems. However, the timely completion of all these individual service invocations are of utmost importance for the overall operation the application tries to complete. Web services middleware seldom contain techniques to achieve execution time predictability. Many of the techniques they employ in achieving higher throughput rates adversely effect the request execution and result in unpredictable execution times. Achieving predictable execution times is important for many reasons. Firstly, consistent and predictable execution times are vital for the successful adoption of web services as an efficient and reliable middleware. Secondly, predictable execution times are important for QoS based service selections in compositions, where composite services may choose between services depending on the execution times they could guarantee. Thirdly, unpredictable execution times will setup clients for failure. We firmly believe that it is better to reject a request with an execution deadline that cannot be guaranteed, rather than accepting the request with the expectation of meeting the perceived execution deadline and being unable to meet it. This would allow a client to look for another service that could meet the perceived

deadline. Finally, predictability of execution is important to many applications that have stringent execution time requirements [Buttazzo, 1997; Natarajan and Zhao, 1992; Stankovic, 1988; Stankovic et al., 1998], such as financial trading systems , manufacturing execution systems, industrial control systems, medical diagnostic systems, avionics and robotics. As a result, such applications have been unable to benefit from the advantages web services are famous for, such as being platform independent, uncomplicated uniform access model and being loosely coupled.

3.2 Problem Statement

Predictability of execution is seldom considered as a design goal in developing web services middleware. On the contrary, they are designed to achieve high levels of throughput [Apache Software Foundation, 2009; Chapell, 2010; Sun Microsystems, 2009]. For instance, requests are accepted for execution unconditionally and executed in a *best-effort* manner. Multiple requests are executed in parallel using processor sharing following the *Thread-pool* concurrency pattern [Graham et al., 2004]. Moreover, executions of all requests are treated with equal priority. Although employing such techniques yield a higher throughput, they become detrimental to the predictability of request execution. For instance, processor sharing leads to an increase in average execution time, proportional to the number of requests being executed in parallel. The execution time of a request becomes highly unpredictable as it varies with the number of requests being executed concurrently at a given time, their execution times and request arrival rates. Moreover, this leads to longer and unpredictable waiting times.

Achieving predictability of execution requires the invocation of a service to be completed within a perceived deadline in a repeatable and a consistent manner. Such a feat is only possible if web services middleware supports execution deadlines for service invocations and have mechanisms of achieving the deadlines. While QoS aspects of service execution has been widely researched, only a few attempts have been made on achieving execution time QoS in web service middleware. However, none of them support execution time deadlines and consider any predictability related parameters such as laxity of a request in its operations.

Given the open environment of the Internet where web services operate in, there is no knowledge about the properties of requests prior to their arrival at the system. Given this unknown nature of requests, giving a guarantee on their execution times is quite a

challenge. Such a feat will only be possible by using appropriate scheduling strategies that has a focus on achieving such execution deadlines.

3.3 Overview of the Solution

Web services middleware can only achieve predictability of execution through purposeful execution of requests with the aim of achieving their execution deadlines. As the first step, we introduce the notion of an execution deadline to each service invocation which will be communicated to the web services middleware by the request, on its arrival at the system. We use real-time scheduling principles in a two step process to achieve these execution deadlines. We assume that every service invocation request has an associated hard real-time deadline that the execution must complete within.

Given the unknown nature of requests and their arrival times, there will be many requests with overlapping deadlines and execution times. Depending on the processing resources available, it will not be feasible to meet the deadlines of all such requests. A proposed analytical model based on real-time scheduling principles, calculates the required processing resources for a request, and confirms whether a deadline of a request could be successfully met. The proposed model contains three main components, namely remaining execution time, processor demand and loading factor, that is used for this purpose. It defines the remaining execution time to indicate the amount of remaining work to be done on a request. The processor demand within a given time period indicates the amount of processing resources required within that time period by one or more active tasks. Finally a loading factor captures the remaining laxity of requests within a time period, considers them together with the processor demand for the same period and calculates a single indicator of whether one or more deadlines may be missed.

An admission control algorithm based on this analytical model is used to decide on the acceptance of every incoming request. The algorithm has two steps. The first step checks whether the execution deadline of the newly arrived task could be met. This is done by calculating the loading factor for the lifespan of the new task. The calculation takes into account already accepted requests that have completion deadlines within the lifespan of the new request. On the assurance that its deadline could be met, the algorithm continues onto the second step where it checks the loading factor between the arrival time of the new request and the deadline of every request already accepted,

completing execution after the lifespan of the new request. This step ensures that the acceptance of the new request will not compromise the deadlines of already accepted tasks. Positive results in both steps of the algorithm will result the new request being accepted for execution and is otherwise rejected. The request selection process of the analytical model and the schedulability check algorithm results in a large range of laxities at the server. This mix of laxities mean that some requests are able to delay their execution without missing their deadlines giving chances to other requests with overlapping lifespans and deadlines, thereby increasing the total number of requests that can be scheduled to successfully meet their deadlines. The second step in using real-time scheduling is in the form of using the earliest deadline first scheduling to execute the accepted requests in the increasing order of their deadlines.

The contributions in this chapter are the laxity based analytical model, admission control algorithm and the deadline based scheduling method that enables web services middleware to achieve predictability of service execution. The uniqueness of the solution is in the use of real-time scheduling techniques that are typically used at design time in closed environments where request properties are known *apriori*. In the proposed solution, they are used at runtime, in a highly dynamic and open environment where request properties are relatively unknown.

The rest of this chapter is organised as follows. In the next section we discuss some of the related work in this area. In Section 3.5, we present the proposed mathematical model and schedulability check algorithm. Next, we present an analytical evaluation of the proposed model and algorithms in Section 3.6) which also gives a theoretical view on how the schedulability check and deadline based scheduling works together in achieving predictability of execution. An empirical evaluation of the solution is provided in Section 6.8 where we measure the predictability gain achieved by our method. Finally we conclude in Section 6.9.

3.4 Related Work

Many existing work related to web service execution could be found in the area of QoS. A common feature that could be observed in many of them is service discovery or composition based on a QoS criteria. Many of them make the assumption that the web services middleware and the underlying infrastructure used, will guarantee perceived QoS levels within a probabilistic measure. Work by Ran S. [Ran, 2003] and Tian M. et

al. [Tian et al., 2003] facilitate the discovery of services based on QoS parameters by incorporating information about QoS levels provided by services in modified web service directories. QoS brokers facilitate the discovery by liaising with the directory service on behalf of the clients, based on its QoS requirement. Liang Q. et al. [Liang et al., 2009] proposes a unified service selection scheme that uses multiple QoS attributes such as execution time, availability and user perceived QoS levels. It carries out the negotiation process based on QoS parameters transparent to the users when services are selected. The work assumes that services will adhere to a probability based guarantee of the QoS parameters they support. Work of Zeng L. et al. [Zeng et al., 2003, 2004] extends this to service compositions where the selection of services for a composite service is based on QoS attributes each service is able to provide. Yet again the guarantee of QoS levels by the middleware is assumed.

While the work mentioned assume QoS levels are guaranteed by the middleware, there are attempts at achieving different levels of quality in the middleware. Work of Sharma A. et al. [Sharma et al., 2003] introduces few methods of differentiation into the processing of requests. Requests are classified into various service classes based on non-functional attributes such as nature of application (i.e. a stock trading service versus a price lookup service), the device being used as a client (i.e. a Personal Computer versus a mobile device) and nature of client (i.e. paying customer request versus a free request). Priorities are assigned to each service class based on these attributes. The arrival rate of each category is monitored and the priorities are dynamically adjusted using a penalty function to reduce starvation. The penalty function penalises a priority on a lower than normal arrival rate and enforces it positively on higher than normal arrivals. The solution achieves some level of differentiation in the throughput of the requests and tries to maintain a pre-defined ratio between the service classes. Similarly, the work of Tien C. M. et al. [Ching-Ming Tien, 2005] classifies requests into service classes based on a pre-defined SLAs between the service provider and clients. Operations are profiled offline and the information obtained is used to calculate the workload required when a service is invoked. A scheduler component in the middleware evaluates the request arrivals and ensures a pre-defined ratio of the service classes, in processing. Although the operations considered in the approach are non-functional properties of the service such as security processing, the same technique could be applied for functional attributes as well. However, the scheduler simply maintains the ratio of the different classes in the number of requests processed, rather than considering the actual execution times

resulted.

The use of admission control in Web Services could also be found in literature. Work of Dyachyuk D. et al. [Dyachuk and Deters, 2007] proposes the use of a proxy between the client and the server with an admission control mechanism that can use different types of scheduling techniques such as First-In-First-Out (FIFO) and Shortest Job First (SJF) to schedule the execution of web service requests. The goal is to achieve control over service execution and prevent the server from reaching overloaded conditions. A similar approach is found in [Elnikety et al., 2004] where a proxy based admission control is used in a three tier web application. The admission control is based yet again on request scheduling although this work focuses more on requests between the application server and the database. FIFO and SJF are techniques used for this purpose and the gateway proxy they introduce prevents overload conditions in the server. A more general approach can be found in [Carlstrom and Rom, 2002] where admission control is used to ensure low session delays by means of a reward function that works on weights associated with different pre-defined stages of processing in the application. Stages represent the discrete functionalities the web application provides and weights are assigned based on the importance of the operation to the user in terms of the delay experienced. [Erradi and Maheshwari, 2005] proposes a QoS-aware middleware similar to an Enterprise Service Bus, which uses priority based differentiation of web services requests. It contains an admission control mechanism as part of its functionality which decides controls the incoming messages from different transport protocols. However, the paper does not provide more details on the scheduling techniques used, except for the prioritisation of messages. The paper also fails to mention the method of prioritisation. While all these works use admission control to prevent overloaded conditions or to control scheduling of requests, none of them uses predictability as the goal for the selection of requests.

Helander J. et al. [Helander and Sigurdsson, 2005] uses SOAP based web services in an embedded real-time environment where web services are used for communication between the components in the system. Patterns of communication, sequence of events, their resource requirements and execution times are known *apriori* due to the embedded nature of the system. Therefore, the execution sequences and the schedules are planned out at design time of the system. A statistical model is used to plan for any variations or possible jitter and resources are over reserved to counter such scenarios. [Mathes et al., 2009a] presents a real-time SOAP engine for industrial automation

which supports all three types of real-time tasks. While there is no clear mention of the scheduling technique used, requests seem to be accepted without being subjected to an admission control check. From these features and the type of evaluation conducted, it can be concluded that this is intended to be used with closed systems, where task properties are known at design time. They propose two methods for finding task properties and favours a profiling approach which is more empirical. The closed nature of these two systems allows planning ahead, thereby reducing the variation in execution times. However, such solutions would not work well with open systems, as in the use of web services over the Internet where request characteristics, arrival patterns and sequences are unknown and unpredictable at design time. Most of the discussed work, maintain a certain ratio of processing between the request classes and achieve perceived execution times in a probabilistic manner. While all attempts achieve some form of differentiation in request processing, none of them can guarantee predictable execution times under any traffic condition. Although [Helander and Sigurdsson, 2005] and [Mathes et al., 2009a] uses real-time scheduling techniques in their solution, the requirement of having all information necessary for scheduling at design time makes it difficult to be used in open systems such as on the Internet due to the unpredictable nature of requests.

3.5 The Proposed Analytical Model and Algorithm

In this section the proposed analytical model and algorithms for achieving predictable execution times are presented. In describing them, we make the assumption that every web service invocation request will specify a deadline that the execution must complete within. It is used by the proposed model and algorithms in deciding on the acceptance of a request and to schedule it for execution.

Real-time scheduling techniques focus on completing the execution of a task within a perceived deadline. They are typically used at design time of a real-time system to work out a execution schedule for tasks that are known to be in the system. Validity of the execution schedule is confirmed through a step known as Schedulability Analysis (discussed under Background in Chapter 2), which is conducted once per schedule at design time of a system. However, the proposed solution uses real-time scheduling in an open environment where request properties are not know prior to their arrival at the system. Given these conditions, the proposed solution will have a dynamic schedule that changes every time a new requested is accepted for execution. As a result, validation of

such a schedule has to be done every time a new request arrives at the system. Similarly, the unknown nature of requests will make it impossible for the system to achieve the deadlines of every request that arrives at the system. Therefore, the proposed solution accepts only requests with achievable deadlines. To cater both these issues, we propose an analytical model for runtime schedulability analysis, that will be conducted for every incoming request.

3.5.1 Analytical Model for Schedulability Analysis

The proposed analytical model aims to achieve two basic functions. Firstly, it defines a set of system parameters that quantifies the required processing resources for a given period of time, considering the active requests within that time period. Secondly, it aims to derive a single value representation that quantifies the possibility of accepting a web service invocation request, that can be used in the decision making process. This single value must represent the possibility to meet the execution deadline of the request as well as its effect on the deadlines of already accepted requests.

First, we define some system parameters to quantify the requirement for processing resources within a given period of time. In a pre-emptive scheduling system, execution of a given request could happen with several pre-emption cycles.

Definition 4 For a given request T_i having n number of pre-emptions, where the start time of each execution is s_n and the end time of each execution is e_n, Total time of the task execution E_i can be considered as,

$$E_i = \sum_{j=1}^{n} (e_j - s_j). \tag{3.5.1}$$

Definition 5 For a given request submitted to the system, with an execution time requirement of C_i, at any given point of time the remaining execution time R_i can be considered as,

$$R_i = C_i - E_i. \tag{3.5.2}$$

When a newly submitted task arrives at the system, the schedulability check is done to ensure it could successfully be scheduled together with the tasks already in the system. The proposed schedulability check calculates the processing requirement of the

new task against the tasks in the system, in a number of cycles. A segregation of already accepted tasks is done, on the basis of whether the deadline of a task lies within the lifetime of the new task or thereafter. The first step is to ensure that the deadline requirement of the new task would be met. For this purpose, only the requests with deadlines within the lifespan of the new request are considered in the calculation.

Let T_{new} be a newly submitted task, with a release time of r_{new} and a deadline of d_{new} and an execution time requirement of C_{new}. Let *P* be the set of tasks already accepted and active in the system, with their deadlines denoted as d_p. We consider the semi-closed interval denoted by the r_{new} and d_{new} as the lifespan of the new request.

With reference to definition 2.2.2, the processor demand within the duration of the newly submitted task can be defined as,

$$h_{[r_{new},d_{new})} = \sum_{r_{new} \leq d_p \leq d_{new}} R_p + C_{new}. \tag{3.5.3}$$

Processor demand ($h_{[r_{new},d_{new})}$) quantifies the processing time requirement within the lifespan of the new task, which includes its own execution time requirement and the remaining execution times of every task in the system that is scheduled to complete within its lifespan. Next we define the term loading factor as a single value indicator of whether the deadline of the new task can be achieved.

With reference to definition 2.2.3, the loading factor within the duration of the newly submitted task can be defined as,

$$u_{[r_{new},d_{new})} = \frac{h_{[r_{new},d_{new})}}{d_{new} - r_{new}} \tag{3.5.4}$$

Loading factor considers the processor demand in its calculation, thereby quantifying the impact of all requests scheduled to complete within the lifespan of the new task. With condition 3.5.4, if the following condition is satisfied, the new task is considered schedulable together with tasks finishing on or before its deadline, with no impact on their deadlines.

$$u_{[r_{new},d_{new})} \leq 1 \tag{3.5.5}$$

With the above condition 3.5.5 satisfied, the task is checked for schedulability with the tasks finishing subsequently. unlike 3.5.3, for this step the processor demand attribute is defined again separately for each task scheduled to complete execution after d_{new}.

Let *Q* be the set of tasks already accepted and active in the system, required to finish after d_{new} (such that, with deadlines after d_{new}). Let *q* be the member of *Q*, with a deadline of d_q up to which the processor demand is calculated for,

$$h_{[r_{new},d_q)} = h_{[r_{new},d_{new})} + \sum_{d_{new} \leq d_i \leq d_q} R_i. \tag{3.5.6}$$

The result of 3.5.3 is used as part of the equation. This represents the processor demand of all tasks finishing on or prior to d_{new} and can be treated as one big task with a release time r_{new} and a deadline of d_{new} respectively. Next, the loading factor for the same duration is calculated.

$$u_{[r_{new},d_q)} = \frac{h_{[r_{new},d_q)}}{d_q - r_{new}} \tag{3.5.7}$$

The loading factor is also calculated on a per task basis for each member of *Q*. Subsequently, the calculated loading factor is compared to be less than or equal to 1, in order for all tasks leading up to *q*, to be satisfied as schedulable.

$$u_{[r_{new},d_q)} \leq 1 \tag{3.5.8}$$

In summary, for a newly submitted task to be accepted to the system, condition 3.5.5 needs to be satisfied for tasks with deadlines on or before d_{new}, subsequently condition 3.5.8 needs to be satisfied, separately for each task with deadlines after d_{new}.

3.5.2 Schedulability Check Algorithm

Using the proposed model for schedulability analysis presented in Section 3.5.1, Algorithm 1 forms the core of our solution. Every incoming web service request is accepted for execution subjected to this schedulability check. This admission control mechanism primarily satisfies two main conditions when accepting a request. Firstly, a request is accepted only if its deadline requirement met. Herein, this is decided by existing requests

Algorithm 1 Schedulability Check

Require: New request N, Queue of Accepted Requests RQ
Ensure: N is accepted or rejected

```
1.  PDW ← 0; PDA ← 0
2.  withinTasksChecked ← false
3.  while RQ has more and withinTasksChecked is false do
4.     nextReq ← RQ.getNextReq
5.     if nextReq.startTime ≥ N.startTime and nextRequest.deadline ≤ N.deadline
       then
6.        PDW ← PDW + nextReq.getRemainingTime
7.     else
8.        if nextReq.deadline ≥ N.deadline then
9.           withinTasksChecked ← true
10.       end if
11.    end if
12. end while
13. PDW ← PDW + N.getRemainingTime
14. LoadingFactor ← PDW / (N.deadline − N.startTime)
15. if LoadingFactor > 1 then
16.    return false
17. end if
18. PDA ← PDW
19. while RQ has more requests do
20.    nextReq ← RQ.getNextReq
21.    PDA ← PDA + nextReq.getRemainingTime
22.    LoadingFactor ← PDA / (nextReq.deadline − N.startTime)
23.    if LoadingFactor > 1 then
24.       return false
25.    end if
26. end while
27. return true
```

in the system that finish within the lifespan of the new request, thereby having earlier deadlines. Secondly, the acceptance of a request must not compromise the deadlines of already accepted requests. The acceptance of a request may result in the delaying or phasing out of the execution of requests with later deadlines than that of the new request. This step ensures the operation would not result in a deadline. Functionality of Algorithm 1 is summarised in Figure 3.1.

We make the assumption that the list of requests already accepted are sorted in the increasing order of their deadlines. At the arrival of a new request, the algorithm considers the deadline requirement of the new arrival (*N*), the list of already accepted requests (*RQ*) and returns whether the new request can be accepted for execution or has

to be rejected. The analytical model will be referred to as (*A.M.*) and the corresponding equations will be cited within the description where applicable.

The schedulability of the new request *N* is checked in two parts. The first part checks whether the deadline requirement of the new request can be fulfilled. Remaining execution times of already accepted requests, with deadlines earlier than that of *N* is considered (Line 5). As the execution of *N* would have to be delayed until their completion, the laxity of *N* is checked against the total remaining execution times of the others (Lines 6-13). This is done by calculating the processor demand within the lifespan of *N* (*A.M.* - 3.5.3), where the remaining execution times of accepted requests are totalled (Line 6) and then the execution time requirement of *N* (Line 13) is added to it. Next the loading factor within the lifespan of the new request (*A.M.* 3.5.4) is calculated (Line 14). The loading factor indicates the ratio between the amount of processing required within a time period and the available processing time. A loading factor of more than 1 (*A.M.* - 3.5.5) will result in *N* being rejected (Lines 15-17). A successful loading factor leads to the second part of the check.

Second part of the schedulability check ensures the acceptance of a new request will not result in any deadline misses of already accepted requests. Hence requests with subsequent deadlines to that of *N*, is considered for this step (Lines 8-10). As there maybe multiple such requests accepted, the acceptance of *N* has a domino-effect on the start of their remaining execution. As a result, the effect of *N*'s execution time requirement on their individual laxities is checked incrementally (Lines 19-26). The list of requests being sorted in the increasing order of their deadlines, aids this process. Considering each request (*nextReq*) with a deadline later than that of *N*, the time period considered is between the start time of *N* and the deadline of *nextReq*. The processor demand for this period is calculated (*A.M.* - 3.5.6) by adding the already calculated processor demand within the lifespan of *N* (Line 18) and the remaining execution time of *nextReq* (Line 21). The loading factor for the

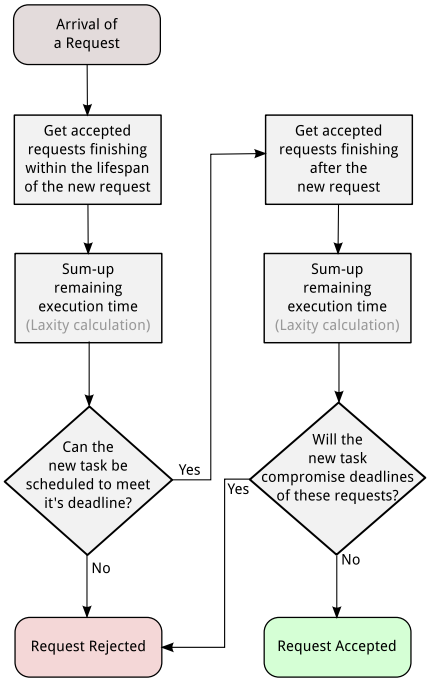

Figure 3.1: Schedulability Check Summarised

time period (*A.M.* - 3.5.7) is calculated next (Line 22) and a result of more than 1 (*A.M.* - 3.5.8) will result in *N* being rejected (Lines 23-25). The check continues on till all requests with subsequent deadlines to that of *N* are considered individually with successful loading factors. The processor demand used in each iteration of the calculation is a cumulative figure carried forward through the algorithm. This ensures that if acceptance of a request results in a possible deadline miss in at least one of the requests already accepted, it is detected as early as possible and further processing is terminated. Requests that get accepted through the schedulability check result in a large range of laxities, thereby enabling more requests to achieve their deadlines.

Complexity Analysis

Next we analyse the complexity of the schedulability check algorithm. We make the assumption that the list of already accepted requests are ordered in the increasing order of their deadlines and are stored in a data structure that allows constant time access to the next request in line returned by the method *getNextReq*.

It can be observed that the algorithm has two *while* loops (Lines 3 and 19). All the statements outside of the two loops (Lines 1-2,13-18 and 28) will be executed once. The condition on the first while loop (Line 3) states the statements within it will be executed as long as there are more requests in *RQ* to consider and the value of *withinTasksChecked* is *false*. A conditional statement within the loop body compares the deadlines of *N* and *nextReq* and sets the value of *withinTasksChecked* to *true* (Lines 8-10) which contributes to the termination of the loop. If this condition holds true, it means any request in *RQ* thereafter has a deadline later than that of *N*. The second *while* loop is set to run until *RQ* is exhausted (Line 19). Note that at this point of the algorithm, *RQ* may have been partially traversed by the first *while* loop, and the traversal will continue on from that point onwards.

The best case execution scenario for the algorithm would be when there are no accepted requests in the system. In this case, the execution of the two while loops will not take place due to *RQ* being empty. However, the statements outside of the loops will still get executed. The worst case execution scenario for the algorithm would be when both the *while* loops are executed the maximum possible times. Given that each loop goes through a portion of the already accepted requests, the maximum number of repetitions both loops could achieve together would be $\frac{N}{2}$ each.

Let n be the number of already accepted requests in the system. Let $T(n)$ be the running time of the algorithm. Let t_1 be the total execution time of all statements within the first *while* loop. Let t_2 be the total execution time of all statements within the second *while* loop. Let t_3 be the total execution time of all statements outside of the two *while* loops. The worst case running time of the algorithm can be estimated as,

$$\begin{aligned} T(n) &= \frac{n}{2}t_1 + \frac{n}{2}t_2 + t_3 \\ T(n) &= \frac{n}{2}(t_1 + t_2) + t_3 \leq n(t_1 + t_2 + t_3) \end{aligned}$$

The above is valid for all $n > 1$. As such, it can be concluded that $T(n)$ has a linear time complexity or is $O(n)$ in the worst case. As the best case scenario would be when there are no previously accepted requests in the system, only the statements outside of the loops would be executed. Therefore it can be concluded that $T(n)$ is in $\Omega(1)$ in the best case.

3.5.3 Deadline Based Scheduling

In the next step of the proposed solution, requests accepted by the schedulability check algorithm are scheduled for execution using the EDF scheduling policy. EDF scheduling mandates the execution of the request with the earliest deadline at a given time. Implementing this scheduling scheme requires the control of all request executions happening within a web services middleware.

Web services middleware achieve *best-effort* processing with the use of multiple worker threads executing in parallel. In the proposed solution, we control the execution of each of these worker threads with the use of an overlaid priority model introduced as part of the solution. A real-time scheduling component introduced into the middleware decides the order of execution based on the deadlines and manages the priorities within the pool of worker threads active at any given time. The proposed solution takes advantage of server hardware with multiple CPU cores or multiple CPUs to increase the throughput of the system. The solution executes multiple requests with the earliest deadlines on separate lanes of execution, equal to the available number of CPU cores or processors, in parallel. Acceptance of a new request would result in a change of priorities and the order of requests being processed in the system, as a whole. As this is an implementation

based feature, a comprehensive discussion on this entire process, implementations of the necessary algorithms and techniques, is presented in Chapter 5.

3.6 Analytical Evaluation

The objective of this section is to provide an analytical evaluation of the proposed model and algorithms for achieving predictability. The evaluation uses a set of tasks arriving at the system. The properties of the tasks have been chosen to evaluate every possible scenario and execution path in the algorithm. The evaluation validates each of the equations in the proposed model (presented in Section 3.5.1) and the corresponding steps in Algorithm 1, thereby confirming its correctness. Moreover, it will validate the scheduling done using the EDF policy.

Each step identifies the arrival of a request at the system and the schedulability check performed on it. If the request is accepted, the real-time scheduler takes a decision on when the request would be executed. Properties of the requests such as their arrival times, execution requirements and Laxities are summarized in the Table 3.1, in the order of arrival. The laxity of a requests is not used directly in the calculations. However, it is considered (indirectly) when the workload (processor demand) that needs to be completed between the lifetime of a task is considered. The corresponding equation in the analytical model is specified as (*A.M. - Eq.*)

Request	Start Time (ms)	Execution Time (ms)	Deadline (ms)	Laxity
T1	0	5	25	20
T2	1	6	19	13
T3	3	3	7	4
T4	4	4	7	3
T5	7	2	3	1
T6	8	7	10	3
T7	9	2	6	4

Table 3.1: Properties of Requests

The example starts off with an empty system. The arrival of the first request T1 is shown in Figure 3.2. As per Table 3.1, T1 has an execution time requirement of 5ms that needs to finish within a deadline of 25ms. In Figure 3.2, the remaining execution time is illustrated using a dotted line while the deadline has been marked using a straight

line. The schedulability check is not carried out for the first arrival.

Request T2 arrives in the system 1ms later (Figure 3.3). By the time which, T1 has executed for 1ms. With its arrival, the schedulability check is performed on T2. As there are no requests in the system with deadlines prior to that of T2, the first part of the schedulability check depicted in Algorithm 1 (Lines 5 - 12) is not applicable. However, rest of the check (Lines 13 - 27) is applied as follows.

Proc. Demand Within	=	(0 + 6)ms	(*A.M.* - 3.5.3, Alg. 1: Line 13)
	=	6ms	
Proc. Demand After	=	(0 + 4)ms	(*A.M.* - 3.5.6, Alg. 1: Line 21)
	=	4ms	
Total Proc. Demand	=	(6 + 4)ms	
	=	10ms	
Loading Factor	=	$\frac{10}{(25-1)}$	(*A.M.* - 3.5.7, Alg. 1: Line 22)
	=	0.4167	
0.4167	$>$	1	(Evaluates to *false* - Accept request)

As visible in the calculation above, the processor demand between the arrival time of T2 and the deadline of T1 is calculated. This constitutes of the remaining execution time of T1 and the execution time requirement of T2. The result is used in calculating the loading factor for the time interval and T2 passes the schedulability check as the loading factor is less than 1. Hence, T2 is accepted for execution. With T2 now being the request

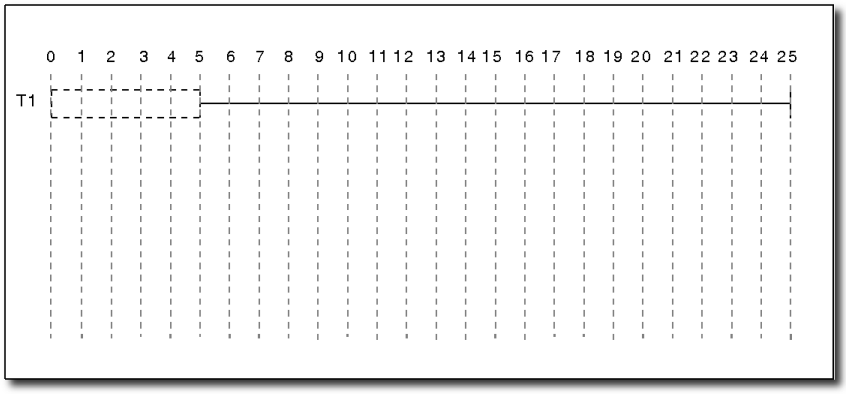

Figure 3.2: Arrival of Request T1

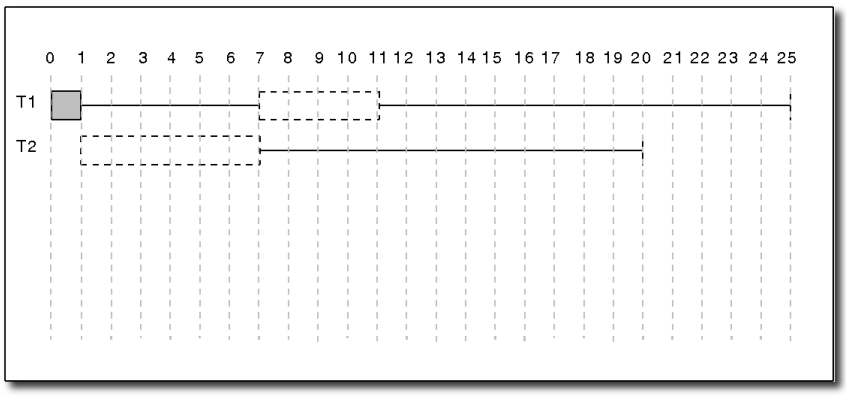

Figure 3.3: Arrival of Request T2

with the earliest deadline, T1 is preempted and T2 is scheduled for immediate execution. The remaining execution of T1 is delayed till execution of T2 finishes. It can clearly be seen that due to its large laxity, T1 can finish within its deadline. In calculating the loading factor the total processor demand is divided by the time period between the arrival time of the new task and the deadline of the existing task under consideration (in this case T1). The deadline of the existing task represents its laxity, which indicates the possibility to accommodate the execution of other tasks with deadlines earlier than it, within its lifespan. A higher laxity in the task considered, results in a lower loading factor.

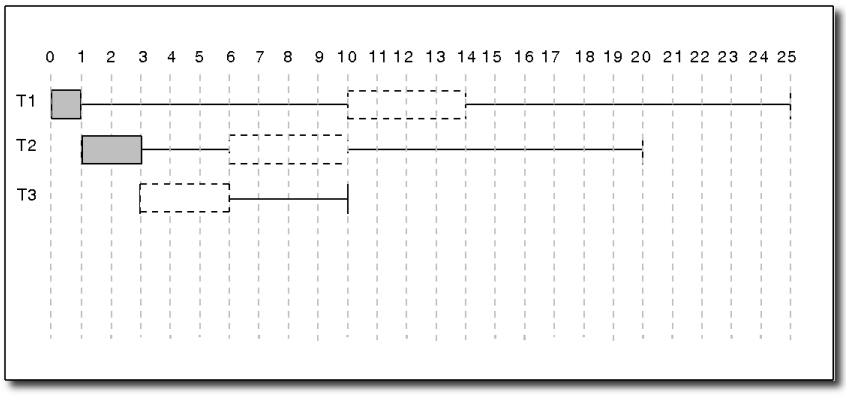

Figure 3.4: Arrival of Request T3

2ms into the execution of T2, T3 arrives at the system (Figure 3.4) and the schedula-

bility check is performed on it. Similarly to T2, there are no requests in the system with deadlines prior to that of T3. Hence, lines 5 - 12 of the algorithm are skipped in performing the schedulability check. For the remainder of the check, as requests T1 and T2 both have deadlines after that of T3, the processor demand and loading factor are calculated up to the deadlines of T1 and T2, separately.

Proc. Demand Within	=	(0 + 4)ms	
	=	4ms	
Proc. Demand up to T2	=	(0 + 4)ms	
	=	4ms	
Total Proc. Demand	=	(4 + 4)ms	(*A.M.* - 3.5.6, Alg. 1: Line 21)
	=	8ms	
Loading Factor	=	$\frac{8}{(20-3)}$	(*A.M.* - 3.5.7, Alg. 1: Line 22)
	=	0.47	
0.47	$>$	1	(Evaluates to *false* - Continue on to next check)

Proc. Demand up to T1	=	(4 + 4)ms	
	=	8ms	
Total Proc. Demand	=	(4 + 8)ms	
	=	12ms	
Loading Factor	=	$\frac{12}{(25-3)}$	(*A.M.* - 3.5.7, Alg. 1: Line 22)
	=	0.545	
0.545	$>$	1	(Evaluates to *false* - Accept request)

We made the assumption that already accepted requests have been sorted in the increasing order of their deadlines. As a result, the check is first performed on the time period between current time and the deadline of T2 and subsequently on the deadline of T1. In calculating the processor demand leading up to the deadline of T2, only the remaining execution time of T2 is considered. T2 having executed for 2ms, has 4ms of execution time left. This results in a total processor demand of 8ms when the execution time requirement of T3 is also considered. When the loading factor is calculated for the time period, it resultant load is less than 1. With no deadline misses leading up to T2, the

processor demand up to the deadline of T1 is calculated. The processor demand calculates to 8ms, due to remaining execution times from both T1 and T2. With a total processor demand of 12ms, due to the execution time requirement of T3, the loading factor leading up to the deadline of T1 builds up to a 0.545, hence the task is accepted as illustrated in Figure 3.4.

With its acceptance, T2 is preempted and T3 is allowed to claim the processor, as it is the task with the earliest deadline. T2 will recommence execution after 4ms followed by T1 recommencing execution after another 4ms. Although the execution of T2 is staged and the re-commencement of T1 further delayed, the larger laxities of T1 and T2, allows T3 to execute within their lifespans while ensuring all 3 requests meeting their respective deadlines.

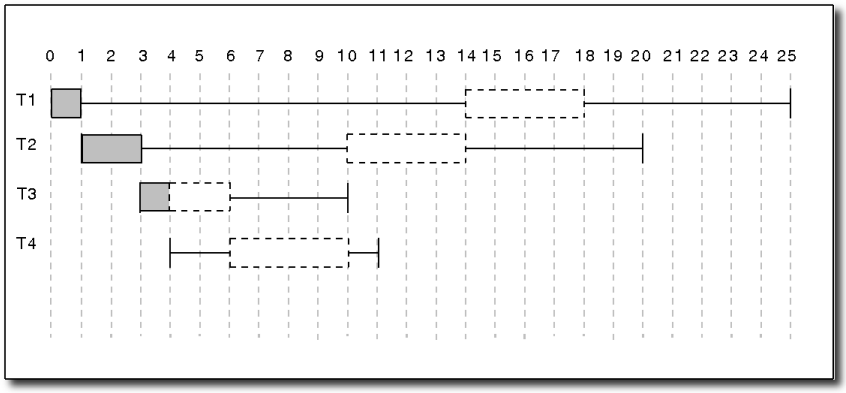

Figure 3.5: Arrival of Request T4

Request T4 arrives at the system 1ms into the execution of T3 (Figure 3.5). The deadline of T4 is 1ms after that of T3 and prior to that of T2 and T1. Therefore, the entire algorithm is applicable for the schedulability check of T4. The check is performed in two parts. The first part calculates the processor demand and loading factor within the duration of the newly arrived request. If the first part of the check is passed, subsequently the processor demand and loading factor between each of the requests with deadlines after T4 is calculated.

Proc. Demand Within	=	(0 + 2)ms	(Alg. 1: Lines 5 - 12)
	=	2ms	
Total Proc. Demand up to T4	=	(2 + 4)ms	(*A.M.* - 3.5.3, Alg. 1: Line 13)
	=	6ms	
Loading Factor	=	$\frac{6}{(11-4)}$	(*A.M.* - 3.5.4, Alg. 1: Line 14)
	=	0.86	
0.86	>	1	(Evaluates to *false*)

As the first part of the check evaluates to *false*, the schedulability check continues on to the second part.

Proc. Demand up to T2	=	(0 + 4)ms	
	=	4ms	
Total Proc. Demand	=	(6 + 4)ms	(*A.M.* - 3.5.6, Alg. 1: Line 21)
	=	10ms	
Loading Factor	=	$\frac{10}{(20-4)}$	(*A.M.* - 3.5.7, Alg. 1: Line 22)
	=	0.625	
0.47	>	1	(Evaluates to *false* - continue to next check)

Proc. Demand up to T1	=	(4 + 4)ms	
	=	8ms	
Total Proc. Demand	=	(6 + 8)ms	
	=	14ms	
Loading Factor	=	$\frac{14}{(25-4)}$	(*A.M.* - 3.5.7, Alg. 1: Line 22)
	=	0.737	
0.737	>	1	(Evaluates to *false* - Accept request)

With T4 having its deadline later than that of T3, the first part of the schedulability check is conducted as T3 finishes within the life span of T4. In calculating the processor demand for the lifespan of T4, the remaining 2ms of execution time of T3 is considered together with the execution time requirement of T4. T4 has a large enough laxity to delay its execution until the remaining execution of T3 is completed within its lifespan.

Therefore, the loading factor results to be less than 1 indicating no deadline misses and the check continues to the second part where tasks finishing after T4 is considered.

Processor demand and loading factor for the time period between the deadline of T4 and the deadline of T2 is first carried out. As the loading factor calculates to be less than 1 the same is calculated for the time period between deadline of T4 and the deadline of T1. Both T2 and T1 has large enough laxities to delay their executions further allowing T4 to finish execution within their lifespans, with no deadline misses.

As T3 still has the earliest deadline, it continues to have the CPU for execution. However, at the completion of T3, the request with the next earliest deadline (T4) will get the CPU for execution. T3 finishes execution at the 6th millisecond since the system started receiving tasks. Therefore, T4 would run from 6 to the 10th millisecond, followed by T2 running from 10th to 14th and T1 running from 14th to the 18th millisecond since its arrival at the system.

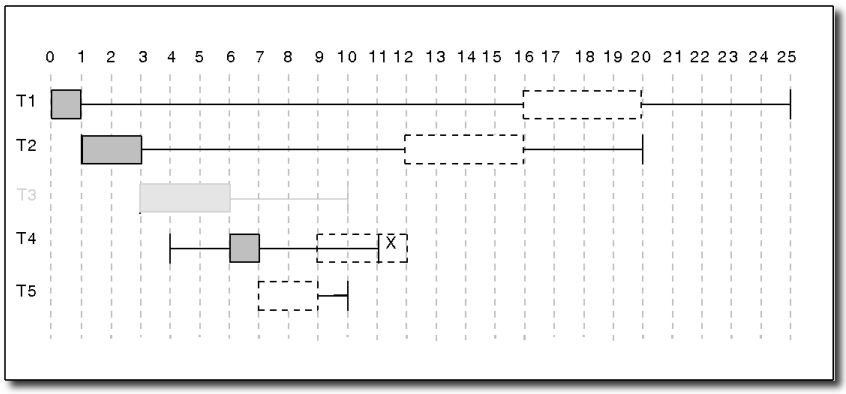

Figure 3.6: Arrival of Request T5

T5 is a relatively small task arriving at the system 1ms into the execution of T4 (Figure 3.6). Moreover, it also has a relatively small laxity. As T5 has a deadline earlier than the rest of the requests, the first part of the schedulability check is skipped.

Proc. Demand Within	=	(0 + 2)ms	
	=	2ms	
Proc. Demand up to T4	=	(0 + 3)ms	
	=	3ms	
Total Proc. Demand	=	(2 + 3)ms	(*A.M.* - 3.5.6, Alg. 1: Line 21)
	=	5ms	
Loading Factor	=	$\frac{5}{(11-7)}$	(*A.M.* - 3.5.7, Alg. 1: Line 22)
	=	1.25	
1.25	$>$	1	(Evaluates to *true* - Reject Request)

Processor demand is first calculated for the time period between the deadline of T5 and the deadline of T4 as the sorted list has T4 being the first request with a deadline after that of T5. The total processor demand calculates up to the remaining execution time of T4 (3ms) and the execution time requirement of T5 (2ms), resulting in 5ms. However, the duration of the time period is just 4ms (11ms - 7ms) in length. As seen above, T4 does not have a large enough laxity to contain the execution of both T5 and its remaining execution time. This results in a loading factor of 1.25 which fails the test. Hence, request T5 has to be rejected.

A loading factor of 1.25 means that if the task was accepted, the total amount of work that needs to be done between the start time of T5 and the deadline of T4 is more than the amount of CPU time that could be allocated for the requests. As T5 would be the task with the earlier deadline, it would gain the CPU continuously till it finishes execution. Thereafter, T4 would be given the CPU as the task with the next earliest deadline. This results in T4 missing its deadline of 7ms from its arrival into the system, which is the 11th millisecond on the timeline. Rejecting T5 ensures that T4 which is an already accepted request can meet its deadline requirement. After the schedulability with T4 fails, the rest of the schedulability check is skipped.

After the rejection of T5, request T6 arrives at the system 2ms into the execution of T4 (Figure 3.7). As T6 has a deadline later than T4, the entire schedulability check is carried out on it.

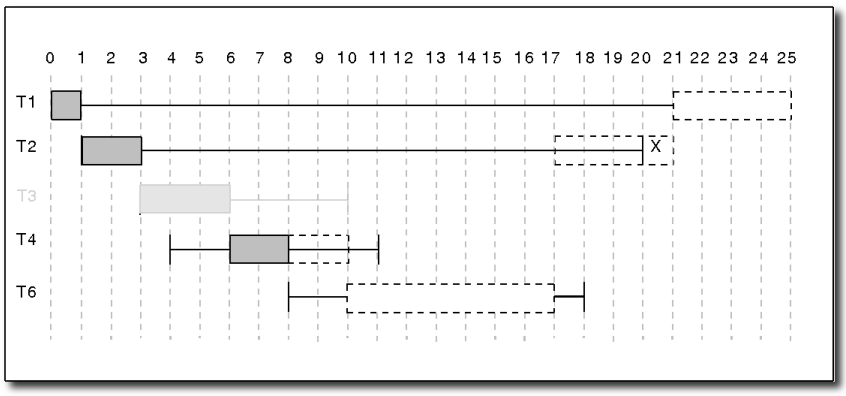

Figure 3.7: Arrival of Request T6

$$\begin{aligned} \text{Proc. Demand Within} &= (0 + 2)\text{ms} \\ &= 2\text{ms} \\ \text{Total Proc. Demand up to T6} &= (2 + 7)\text{ms} \\ &= 9\text{ms} \\ \text{Loading Factor} &= \frac{9}{(18-8)} \\ &= 0.9 \\ 0.9 &> 1 \qquad \text{(Evaluates to } \textit{false}\text{)} \end{aligned}$$

As the first part of the check evaluates to *false*, the schedulability check continues on to the second part.

$$\begin{aligned} \text{Proc. Demand up to T2} &= (0 + 4)\text{ms} \\ &= 4\text{ms} \\ \text{Total Proc. Demand} &= (9 + 4)\text{ms} \\ &= 13\text{ms} \\ \text{Loading Factor} &= \frac{13}{(20-8)} \\ &= 1.083 \\ 1.083 &> 1 \qquad \text{(Evaluates to } \textit{true}\text{ - Reject Request)} \end{aligned}$$

Unlike T5, T6 having a deadline later than T4 would need to have a laxity that could make way for the remaining execution of T4, without missing its deadline. A loading factor of 0.9 within the lifespan of T6 means that it could be successfully scheduled to

meet its deadline while T4 is also completed within its deadline. In the second part of the schedulability check, the processor demand for the time period between the deadline of T6 and that of T2 is calculated. At this point of time, the laxity of T2 is not adequate to contain the execution of T6 within its lifespan as it already phased its execution making way for T3 and T4. The resultant processor demand and loading factor results in 1.08% of CPU utilization. This leads to request T6 being rejected.

Although T6 could be scheduled to meet its deadline while it makes way for the remaining execution time of T4, accepting it for execution would require the execution of T2 and T1 being further delayed. However, this results in T2 missing its deadline by 1ms, although T1 would still be able to finish within its deadline due to having a larger laxity. The rejection of T6 prevents already accepted tasks missing their deadlines, if it was accepted.

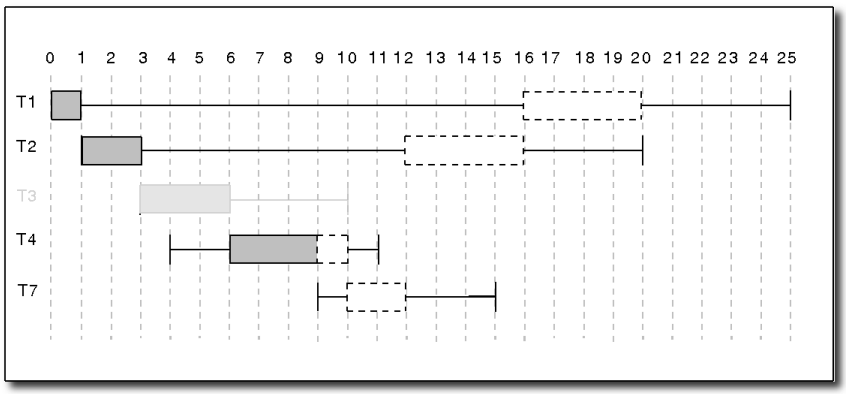

Figure 3.8: Arrival of Request T7

Request T7 arrives at the system, 3ms into the execution of T4 (Figure 3.8). Having a deadline later than that of T4, the entire schedulability check is applicable.

$$
\begin{aligned}
\text{Proc. Demand Within} &= (0 + 1)\text{ms} \\
&= 1\text{ms} \\
\text{Total Proc. Demand up to T6} &= (1 + 2)\text{ms} \\
&= 3\text{ms} \\
\text{Loading Factor} &= \tfrac{3}{(15-9)} \\
&= 0.5 \\
0.5 &> 1 \qquad \text{(Evaluates to } \textit{false}\text{)}
\end{aligned}
$$

As the first part of the check evaluates to *false*, the schedulability check continues on to the second part.

$$
\begin{aligned}
\text{Proc. Demand up to T2} &= (0 + 4)\text{ms} \\
&= 4\text{ms} \\
\text{Total Proc. Demand} &= (3 + 4)\text{ms} \\
&= 7\text{ms} \\
\text{Loading Factor} &= \tfrac{7}{(20-9)} \\
&= 0.636 \\
0.636 &> 1 \qquad \text{(Evaluates to } \textit{false}\text{ - continue to next check)}
\end{aligned}
$$

$$
\begin{aligned}
\text{Proc. Demand up to T1} &= (4 + 4)\text{ms} \\
&= 8\text{ms} \\
\text{Total Proc. Demand} &= (7 + 8)\text{ms} \\
&= 15\text{ms} \\
\text{Loading Factor} &= \tfrac{15}{(25-9)} \\
&= 0.9375 \\
0.9375 &> 1 \qquad \text{(Evaluates to } \textit{false}\text{ - Request accepted)}
\end{aligned}
$$

With T4 having only 1ms of execution time remaining, the processor demand between the lifespan of the newly arrived T7 accumulates up to a small 3ms. T7 has a large enough laxity to comfortably support the execution of T4 and its own within its lifespan. Therefore, the first part of the schedulability check results positive. Next, the

schedulability of T7 is checked with T2 and T1. Both these tasks have enough laxity to further delay their execution making way for T7 to finish within their lifetimes. The resultant loading factors of 0.636 and 0.9375, indicates that the request T7 can be successfully scheduled with each of the already accepted tasks executing in the system.

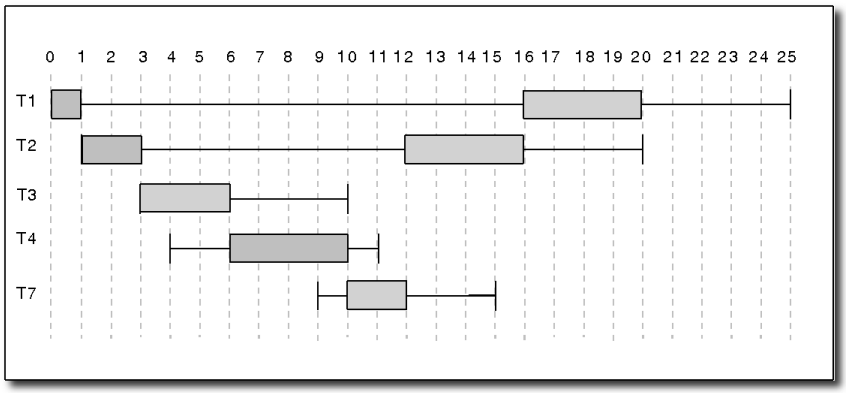

Figure 3.9: Completed Schedule of all accepted Requests

Figure 3.9 illustrates the completed schedule for all accepted requests. Although the execution of some tasks were phased out, all accepted tasks were able to meet their deadlines successfully. Furthermore, it is evident that laxity and the arrival time plays a part in the acceptance of a request. Rejection of requests with deadlines and execution time requirements that cannot be accommodated within the available CPU time, ensures that already accepted requests in the system would not be penalised for their execution.

3.7 Implementation

To empirically evaluate the effectiveness of the proposed solution, aforementioned algorithms and techniques are implemented in an existing web services middleware product, namely Apache Axis2 [Apache Software Foundation, 2009]. The notion of a deadline was introduced to each web service request. This property can be specified for each service invocation. It is passed onto the middleware by using SOAP headers. The core functionality of Axis2 was modified to retrieve the deadline from the SOAP header and conduct the schedulability check prior to the acceptance of a request for execution.

The proposed functionality of these algorithms were facilitated by a few techniques

introduced in the implementation process. The implementation of the EDF policy required a change to the *best-effort* nature of execution in Axis2. Firstly, all thread-pools used by Axis2 were replaced with custom built real-time thread-pools. A newly introduced real-time scheduler component enforces the EDF policy on the executions in Axis2 by having fine-grain control over the worker threads in every pool. It uses a newly introduced priority model to differentiate request processing and control the executions by suspending and resuming execution of individual worker threads at will. These implementation features were further facilitated at the system level by the use of real-time development platforms and operating systems in the lower layers of the system.

The use of EDF scheduling mandates the sequential execution of requests. However, in order to achieve an acceptable level of throughput, the real-time scheduler uses multiple lanes of execution. The number of lanes are configured to be one less than the number of CPU cores available on the server. Therefore on a server with n CPU cores, the tasks with the $n-1$ earliest deadlines at any given time will be executed. An extensive discussion on the implementation of RT-Axis2 is presented in chapter 5.

3.8 Empirical Evaluation of the Proposed Solution

Next we evaluate the level of predictability achieved by the use of the proposed schedulability check and deadline based scheduling in web services middleware. For this, we compare the predictability gain by the enhancements made to RT-Axis2 with an unmodified Axis2 deployment. There were no other techniques or algorithms used for achieving predictability in web services that we could compare with. To measure the level of predictability achieved by the proposed algorithms, the implemented system must be compared with similar web services middleware implemented using a similar development platform. At the time of conducting this research there was no other open source web services middleware that was developed using Java development platform. Therefore, the only feasible comparison was with an unmodified version of Axis2. Moreover, to measure the predictability of execution or performance of web service middleware, there is no widely accepted or used data set available. Therefore, the systems were exposed to request streams created by us using a custom traffic generator, through which the task size, inter arrival rates and deadlines can all be varied accordingly. To create a worst-case scenario where the request properties are highly variable, we used uniformly distributed task sizes, arrival rates and deadline rates and considered all requests to have

hard real-time deadlines, in these experiments.

3.8.1 Experimental Setup

Due to the dynamic environment web services operate in, the solution must be evaluated in highly variable traffic conditions. Although in reality we could expect our solution to be exposed to mixed stream of requests with only a portion of them having hard deadlines, the evaluation is conducted for the extreme case of the entire request stream having hard deadline requirements. To represent the highly variable task sizes and different arrival rates that exists in the real world, we use uniformly distributed task size and inter-arrival times in our experiments. We use a web service that allows us to create different sized workloads on the server with the input parameters used. To measure the effectiveness of the enhancements, the implementation is compared against an unmodified version of Axis2. The metrics used for the comparison are the percentage of requests accepted for execution and the percentage of deadlines met out of the accepted requests. Whilst the unmodified version does not employ any admission control mechanisms, it rejects requests in overloaded conditions.

Figure 3.10 illustrates the experiment setup. The following hardware and software were used as the test environment. Both the enhanced version and the unmodified version of Axis2 were deployed on servers with dual Intel Core 2 Duo 3.4 GHz processors (4 cores in total) with 4 Gigabytes of RAM, Gigabit Ethernet port running Sun Solaris 10 update 05/08 with RTSJ version 2.2. RT-Axis2 is configured with 3 lanes of execution with 100 worker threads for the stand alone deployment. Realistic values were used as deadlines in the experimental evaluations. For this purpose, we profiled the web service for a range of input parameters and derived a functional relationship between the input values and the resultant execution time. The deadline for each task was calculated as a multiplication of the execution time by a random value ranging from 1.5 to 10.

Five client machines are used to generate requests to the server. Ubuntu Linux version 8.04 with the Linux Real-time kernel 2.6.21 was used as the operating system. Although the performance measurements were done only on the server side, it was decided to make use of a real-time operating system and develop the request generating software using real-time development libraries to ensure that the request generation process happens in a timely and uninterrupted manner. This was primarily to ensure the accuracy of the arrival rates the tasks are generated at. A controller machine with the

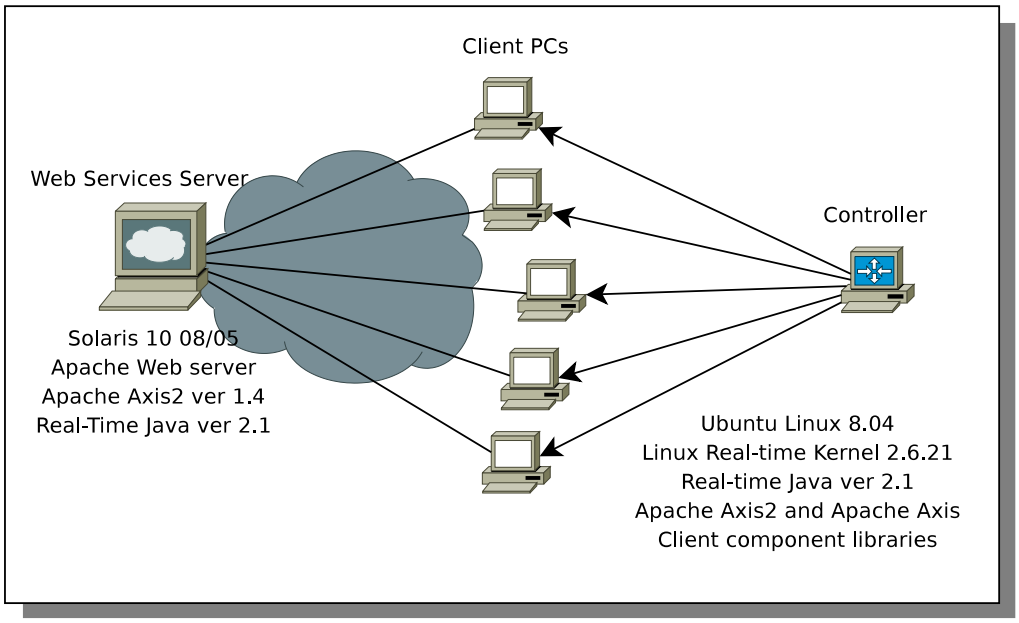

Figure 3.10: Hardware and Software Setup

same hardware and software configuration is used additionally. It controls the entire experiment by deciding the size of the task and the time a request is generated. Moreover, it decides from which client the request is generated from.

3.8.2 Measurement of Predictability in Service Execution

Table 3.2 and Figure 3.11 summarises the comparison between RT-Axis2 and the unmodified version of Axis2. For each setup the first column shows the percentage of requests accepted by the schedulability check and the next column contains the percentage of deadlines met off the percentage of accepted requests. Due to the unconditional acceptance of requests, unmodified Axis2 surpasses RT-Axis2 in the percentage of requests accepted for execution. The unmodified version accepts between 29% - 100% of requests in the given scenarios, while RT-Axis2 accepts between 18% - 96.7% of the requests.The schedulability check in RT-Axis2 finds less requests accepted due to their laxity consideration and the deadline requirement. Request rejections in Axis2 is caused by request time-outs after the system becomes unresponsive due to being overloaded with requests.

When the deadline achievement rate is considered, Axis2 is only able to achieve be-

	Unmod. Axis2		***RT-Axis2***	
Inter-arrival times (sec)	% Acc.	% D. Met off % Acc.	% Acc.	% D. Met off % Acc.
0.25 - 2	100	36.2	96.7	100
0.25 - 1	62.4	18.3	58.6	100
0.1 - 0.5	55.1	9.1	30.7	99.7
0.1 - 0.25	28.7	8.8	18.1	96.7

Table 3.2: Performance Comparison of Unmodified Axis2 vs. RT-Axis2

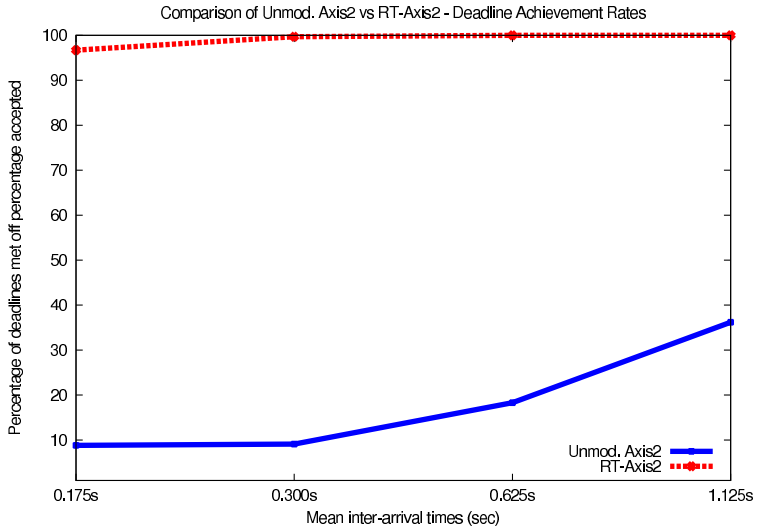

Figure 3.11: Axis2 and RT-Axis2 - Deadline Achievement Rates

tween 9% to 36% of the deadlines from the requests accepted, under experimental conditions. However, RT-Axis2 achieves more than 96% of the deadlines from the accepted requests in all the experiment runs. Due to the *best-effort* nature of request execution, Axis2 results in unprecedented execution times. This can clearly be seen in the top two graphs of Figure 3.12 which shows the median execution times resulting from both systems. This phenomenon leads to majority of the deadlines being missed in unmodified Axis2. The schedulability check prevents RT-Axis2 from having such overload conditions and thereby prevents any impact on the execution of accepted requests. Together with deadline based scheduling, RT-Axis2 achieves more than 96% of the deadlines being met at all times, outperforming Axis2 in the evaluations. Comparing resultant execution times in Figure 3.13, it can clearly be seen that the range of values achieved by Axis2 is far greater compared to the range achieved by RT-Axis2. This is a clear example of the unpredictable nature of *best-effort* execution in such web services middleware. Furthermore, the two graphs in the second row shows the resultant execution times sorted by the task size, it can clearly be seen that the fluctuation of execution times are far greater for large task sizes. Whilst some fluctuation exists even in RT-

Axis2 execution times, they are smaller and are controlled delays based on the laxity of a request.

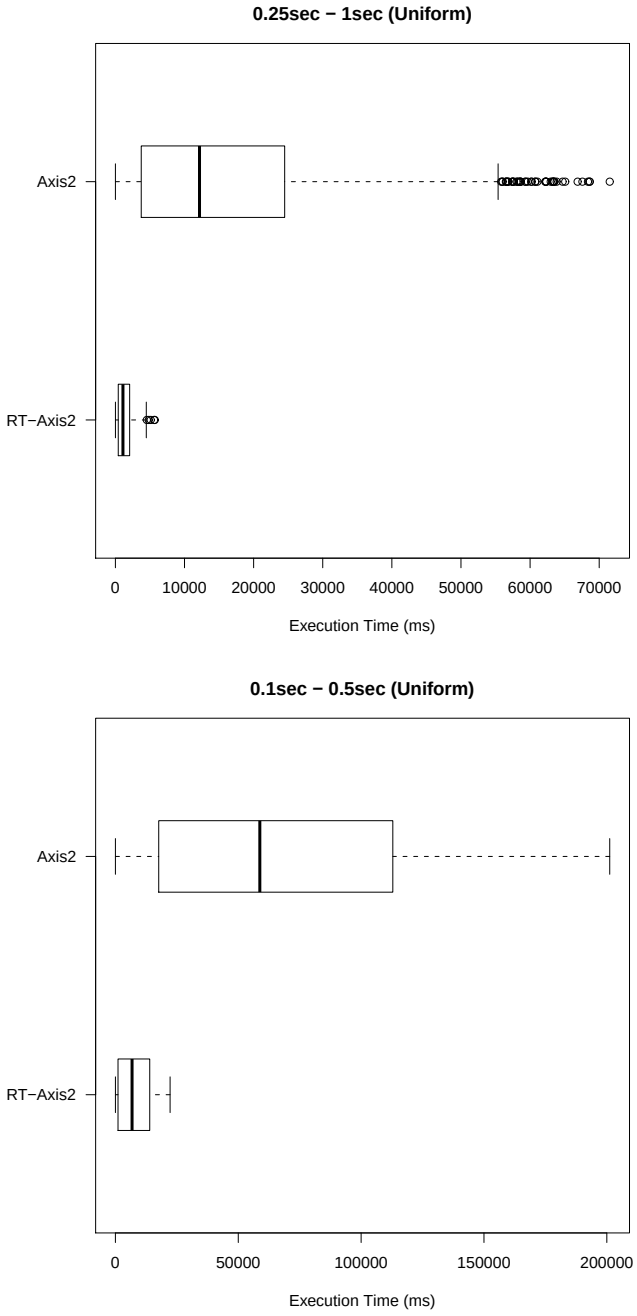

Figure 3.12: Range of Resultant Execution Times

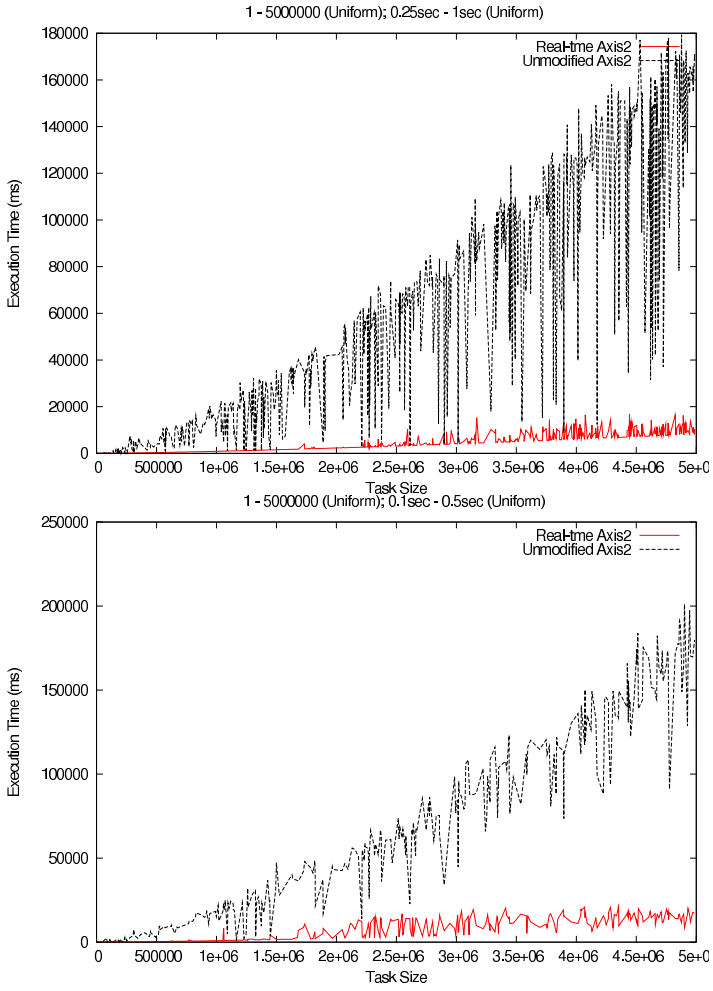

Figure 3.13: Execution Time Variability

3.8.3 Impact of Request Arrival Rate

As the experiment setup is exposed to decreasing inter-arrival times, requests arrive at the system far more rapidly and a decrease can be observed in the percentage of requests accepted. This leads to overloaded conditions in unmodified Axis2 that results in requests being dropped. Moreover, due to the *best-effort* nature of request execution, unmodified Axis2 results in highly variable execution times as seen in Figure 3.13. The saturation of processing resources in Axis2 leads to it reaching the maximum processing capacity and the rejection of subsequent requests. RT-Axis2 is prevented in reaching such conditions by the admission control mechanism we introduced. As this is a common phenomenon with such dynamic environments, having such preventive measures is of paramount importance for achieving predictability of execution.

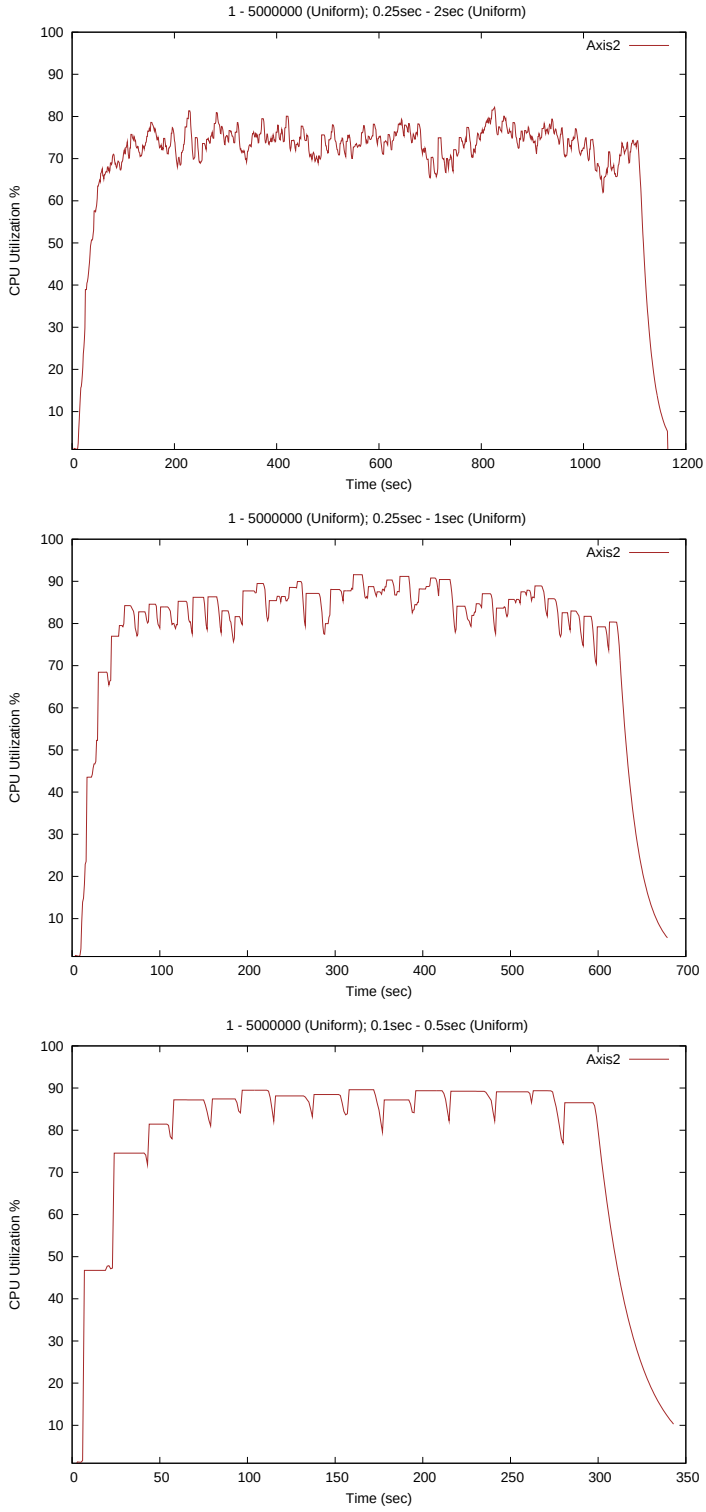

Figure 3.14: CPU Utilisation Levels

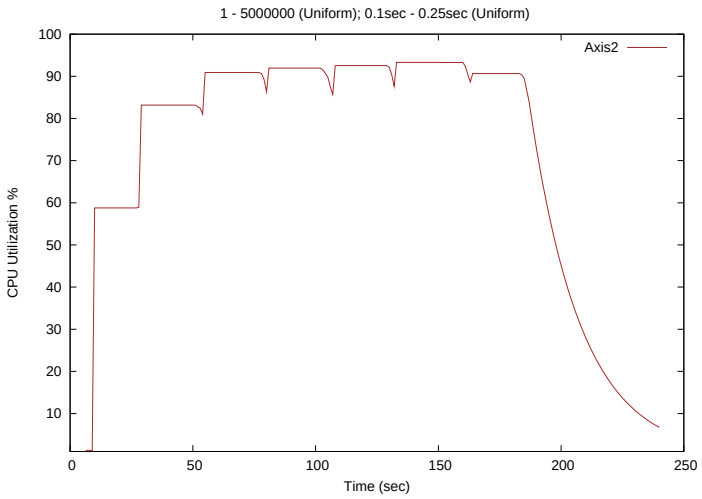

Figure 3.15: CPU Utilisation Levels

3.8.4 Request Processing

While Axis2 rejects requests due to the saturation of processing resources, RT-Axis2 rejects requests through the laxity based schedulability check. As this admission control mechanism prevents the system reaching overload conditions, it will be worth investigating the CPU utilisation at RT-Axis2. Figure 3.14 and 3.15 contains the CPU utilisation for each of the runs. It can be observed that RT-Axis2 achieves nearly 90% of utilisation with increased request rejections. Due to the laxity based selective acceptance of the schedulability check, the CPU is prevented from reaching 100% utilisation thereby possibly leading to overloaded conditions.

3.8.5 Laxity Based Request Selection

From the previous discussions it is evident that enhancements made to RT-Axis2 and RT-Synapse results in conditional acceptance of requests, based on their laxities. The introduced schedulability check works by trying to match the laxity of a target request with the already accepted requests that overlaps with its lifespan in the system. A request is accepted based on the compatibility of its laxity with that of already accepted requests, depending on the processor demand within its lifespan in the system. As illustrated in the sample scenario (Section 3.6), a request with a larger laxity will allow many other requests to be scheduled within its lifespan and a smaller laxity will require the request be scheduled together with other tasks with higher laxities. The nature of

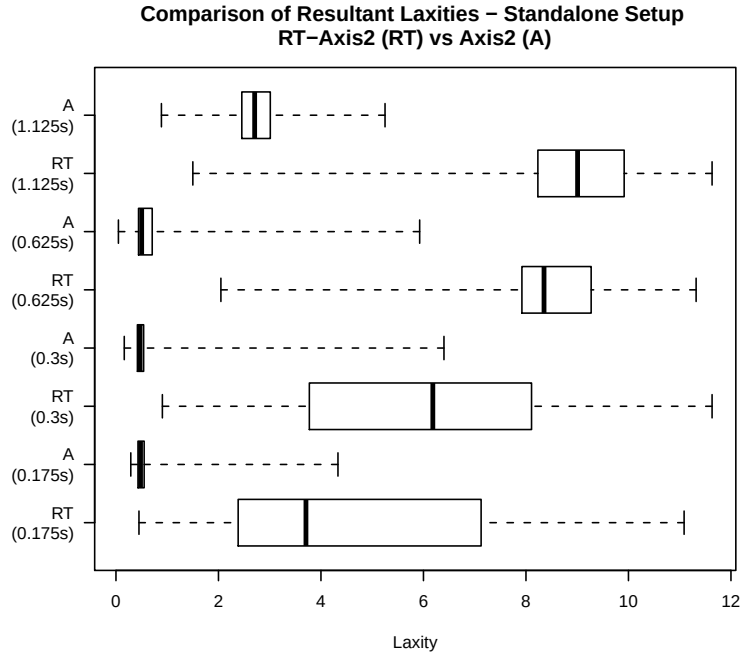

Figure 3.16: Comparison of Resultant Laxities

this selection process eventually results in a wider range of laxities at a server in any given time period.

Recall that laxity indicates the ability to delay the execution of a request, while still meeting its deadline requirement. It is usually indicated as a ratio between the deadline and the execution time of a request. Figure 3.16 visualises the range of laxities resulted at each server for Axis2 (A) and RT-Axis2 (RT). Recall that the *best-effort* nature of Axis2 executes as many requests as possible in parallel and leads to overload conditions and deadlines misses. These conditions gets worse with short inter-arrival times as seen on Table 3.2. In every run, many of the initial requests handled by Axis2 get executed very quickly as the competition for CPU is less, till more requests arrive. This results in lower execution times that contributes to higher laxity values. However, as the number of requests increase the *best-effort* processing overloads the server and many of the requests being executed sharing the processor result in execution times past their deadline requirement. With less than 36.2% of the requests meeting their deadlines, majority of the requests in every Axis2 run results in laxity values less than 1. As visible on the graph, the median laxity value gets lower with the increasing arrival rate.

Similarly, the median laxity value decreases with higher arrival rates for RT-Axis2.

However, the selection of requests by the schedulability check results in a range of laxities at the server. Finding a request with a complementing laxity schedulable with existing requests, results in this phenomenon which can clearly be observed in the figure above. The runs with slow request arrivals result in a higher median value, and it decreases with increasing request arrivals although still resulting in a large variety of values. This contributes towards requests meeting their deadline requirement even in conditions with high task arrivals.

3.8.6 Throughput Comparison

Next we compared the two systems on the throughput rates achieved. Axis2 is designed to achieve good throughput rates in normal conditions. The enhancements made to RT-Axis2 will have a negative impact on the throughput levels achieved by the middleware. However, Axis2 has no mechanisms to prevent system overloads in high traffic conditions, whereas the admission control mechanism in RT-Axis2 prevents it from reaching such a state.

For this discussion we define throughput to be the number of requests processed by a server in a given unit of time. Herein, for the unmodified configurations we consider a request that is executed successfully as a processed request, as there is no differentiation enforced. However, for the enhanced configurations, any request rejected or executed successfully are considered as a processed request. For a rejection, a request needs to be processed by the server up to the completion of the schedulability check using the deadline information fetched. Therefore, this processing qualifies the request to be considered for throughput calculations. Throughput of a server can be mainly affected by three parameters. Firstly, the software by design may have certain features that maximises request processing. Secondly, the processing capability of the software maybe limited by the hardware configuration it is hosted in. Thirdly, request arrivals will have an effect on the resultant throughput of the server.

	Unmod. Axis2	***RT-Axis2***	
Mean inter-arrival time	Throughput (sec^{-1})	Throughput (sec^{-1})	Throughput (excl. rejected)
1.125s (Low)	0.98	0.91	0.88
0.625s	0.83	1.62	0.95
0.300s	0.72	3.40	1.04
0.175s (High)	0.69	5.64	1.02

Table 3.3: Throughput Comparison of Unmodified Axis2 vs. RT-Axis2

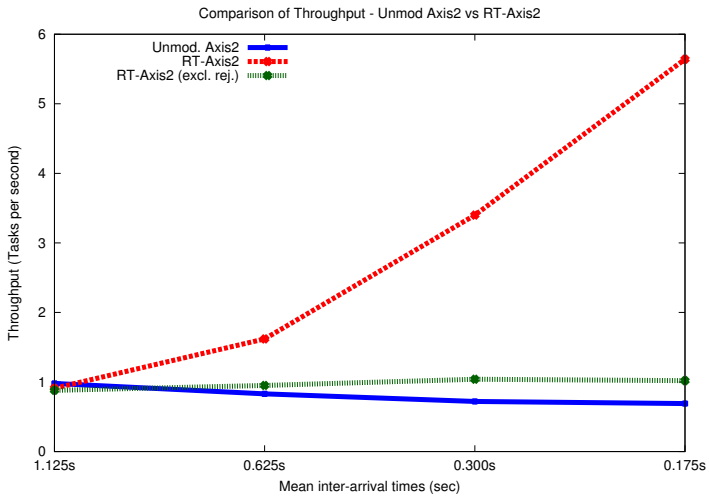

Figure 3.17: Throughput Comparison

Table 3.3 contains throughput values measured (requests per second) for unmodified Axis2 and RT-Axis2 under different request arrival rates, for each scenario discussed earlier. Figure 3.17 summarises the results graphically. The second column for RT-Axis2 contains throughput measured without considering requests rejected. Axis2 is configured by default with 25 worker threads pre-created at start-up and the ability to create up to 150 worker threads when the request queue is filled up. Practically, it is possible for all 150 threads to be in execution sharing the processor at any given time as there are no differentiation or control over how threads execute. With the highest mean inter-arrival time (1.125s), Axis2 records better throughput values compared to RT-Axis2. With increasing arrival rates, Axis2 records decreasing throughput values. The *best-effort* nature of Axis2 contributes to it being overloaded in quick-time and the system being unresponsive, resulting long delays in request completions. Moreover, incoming requests drop out due to unresponsiveness of the system. This condition increases with request arrival rates.

The enhancements made to RT-Axis2 enables control over the execution of worker threads. The configuration of 3 execution lanes and the functionality of the real-time scheduler component, restricts only 3 threads to be in execution at any given time. There maybe up to 100 worker threads pre-created, ready to be used for request execution or with assigned requests with later deadlines. Their use of the CPU is controlled by the scheduler using lower priorities. In the lowest request arrival configuration, RT-Axis2 achieves a marginally lower throughput value. As request arrival rates are increased, the throughput of the system also increases accordingly. Although request traffic increases,

the schedulability check in RT-Axis2 prevents the system from being overloaded. However, a side-effect of this is the increased amount of rejections as observed in Table 3.2. The processing of a request up to the completion of the schedulability check, is comparatively quicker than the intended service execution. A rejected request will incur only this portion of processing. As a result, the throughput recorded in high traffic conditions comprise of a considerable amount of rejected requests. It is clearly observed in the respective secondary throughput values calculated, excluding the rejections. At the smallest mean inter-arrival time (0.175s), the ratio of accepted to rejected requests is around 1:4.

3.8.7 Discussion

Through the empirical evaluation we tried to ascertain the validity of the guidelines provided and the enhancements made accordingly, to web services middleware for achieving predictability of service execution. The empirical results confirm that predictability of execution could certainly be achieved with the suggestions made. The conditions tested for were worst case scenarios of high request arrival rates, all requests having hard deadline requirements and highly variable task size distributions. Unmodified Axis2 the enhanced system was tested against, unconditionally accepted requests for execution, that resulted in higher acceptance rates. However, this led it into overload conditions which resulted in the rejection of requests, as non-responsiveness of the system led to request time-outs. Moreover, this also led to high rates of deadline misses, 58.2% being the highest achieved and at times being less than 10% . Although the RT-Axis2 resulted in lower acceptance levels, it excelled in achieving more than 90% of the deadlines in every scenario tested for.

With the empirical results obtained, it is clear that RT-Axis2 outperforms the unmodified version securing at least 97% of the deadlines while maintaining comparable request acceptance rates. The unconditional acceptance of requests in the unmodified version together with the *best-effort* nature of request execution, works well in scenarios where completion time limit is of less importance. With the thread-pools in use executing as many requests as possible in parallel, their completion times increase with the number of requests being executed, where the maximum number of worker threads serve as an upper bound. As seen on Figure 3.12 and 3.13, the resulting longer execution times contribute to the number of deadlines missed. Moreover, this leads to very high utili-

sation levels, that could create overload conditions, which result in requests timing out due to unresponsiveness of the server. Conditional acceptance of requests based on laxity in RT-Axis2 ensures that a request can be scheduled together with already accepted requests whilst ensuring all deadlines are met. This is further facilitated by the deadline based scheduling used for request execution. Supported by the priority model introduced, the features available in the development platform and the OS, predictability in execution is achieved successfully. With increased arrival rates, more requests compete for the same window of time. In such scenarios, the schedulability check may reject more number of requests. The rejections could be reduced with the use of a cluster based setup we present in chapter 4 where more than one server is used for request execution. Moreover, another approach would be to consider a rejected requests after a certain time window if its deadline has not been expired. This is discussed further under future work in the conclusion chapter 7.

The role of laxity in achieving predictability and its importance can be observed in the results discussed in the laxity comparison. While *best-effort* processor sharing execution is ideal for common processing tasks, ensuring predictability mandates a suitable method of admission control that contributes towards the goal. Request selection based on laxity gives an assurance of meeting a request deadline even prior to its acceptance for execution. The wide range of laxities achieved by the selection process ensures that requests with complementing laxities execute successfully within a given window of time.

The throughput achieved by the RT-Axis2 indicates that its performance is comparable with the unmodified version, in low traffic conditions. Although the RT-Axis2 outperforms Axis2 in high traffic conditions the higher throughput values are largely contributed more by the request rejections. However, when throughput is calculated excluding the rejections both configurations still achieve acceptable throughput rates with resilience to high traffic conditions, contributed by the admission control mechanism. While Axis2 succumb to system overloads, it is bound to perform better than RT-Axis2 in favourable conditions. Therefore, RT-Axis2 can only be considered resilient to high traffic conditions. Considering it to have better throughput values under normal conditions would be an unfair assessment on unmodified Axis2.

3.9 Summary

This chapter, presented means of using real-time scheduling principles to achieve predictability of execution in web services middleware. Due to the highly dynamic and unknown nature of web service requests, the need of an admission control mechanism in selecting requests for execution was discussed. The presented mathematical model enables schedulability analysis at run time using real-time scheduling principles. All web service requests arriving at a web service middleware is subjected to a schedulability check based on the model. Requests are selected for execution based on their laxity property, which indicates the ability to delay or phase out their execution without missing a designated deadline. By selecting complementing laxities, the schedulability check results in a large range of laxities at a server, thereby enabling more requests to be scheduled together while meeting their execution deadlines.

Deadline based execution of requests by the middleware further ensures attaining the deadlines. Requests are executed in the increasing order of their deadlines and due to the aperiodic nature of web service requests, using EDF policy enables dynamic changes to the schedule while achieving schedulable bound of 100%. The popular Axis2 web services middleware is enhanced with these features and the resultant RT-Axis2 is evaluated against the unmodified version to measure the predictability gain. With the enhancements, RT-Axis2 is able to achieve more than 96% of the deadlines while accepting comparable amount of requests as the unmodified version. Unlike in Axis2, these enhancements protects the middleware from reaching overloaded conditions high traffic conditions.

Nevertheless, the amount of requests rejected by the schedulability check is still a concern. With the decreasing hardware costs, a way to reduce the rejection of requests is to have multiple servers hosting web services. In the next chapter we extend the techniques presented to a cluster or servers hosting web services.

Chapter 4

Predictability of Execution in Web Services Clusters†

The use of cluster servers to host and deliver web services is an effective way of catering the growing demands of popular services. Clusters with replicated content are equipped to achieve better response times, better handle of increased loads with the added advantage of increased availability. Herein, the request dispatching algorithm used, has a direct impact on the performance of the cluster. Depending on the objective of the dispatching technique, requests maybe distributed to balance the resultant load among cluster members or be unbalanced to achieve some level of differentiation for the clients. However, consistently achieving predictable execution times for a service is seldom considered as a dispatching goal. This chapter presents four request dispatching algorithms based on real-time scheduling principles (namely RT-RoundRobin, RT-ClassBased, RT-LaxityBased and RT-Sequential) that enable clusters hosting web services to achieve predictability in service execution. The proposed algorithms achieve predictability of execution by incorporating properties such as execution deadline and laxity of a request into the dispatching decisions. This is achieved by conducting schedulability analysis as part of their functionality. Once a request is matched with an executor based on the dispatching policy, the schedulability check considers the laxity of the request and checks with the executor on the possibility of achieving the requested deadline. This additional step of laxity based schedulability check results in a

† Preliminary versions of the work presented in this chapter have been previously published in [Gamini Abhaya et al., 2010a,b, 2012].

wider range of laxities at each executor in the cluster. By maximising the range, it allows more requests with overlapping executions to be scheduled together. RT-LaxityBased algorithm, takes this process further by further ensuring that requests with the same laxity value do not get assigned to the same executor consecutively. This additional step of using laxity enables it to achieve on average 4% more deadlines than the other algorithms. The algorithms are compared with common dispatching techniques such as Round-Robin and Class-based dispatching to measure the predictability gain they achieve. The empirical results show the proposed algorithms outperform the others by meeting at least 95% of the deadlines compared to less than 10% by the others, while maintaining acceptable throughput rates in high traffic conditions.

4.1 Motivation

The Internet has witnessed a growth of web services usage in the recent years. To alleviate performance bottlenecks that may arise from the growing demand, a common solution is the use of clusters in hosting web services. Clusters work by spreading out requests among replica servers based on some pre-defined scheme. This act of balancing the load to gain performance is effective in achieving improved response times and increasing availability of the services.

The most important aspect of a cluster of servers hosting any type of content, is its method of request distribution among the cluster members [Cardellini et al., 2002]. The dispatching algorithm controls the nature of requests each executor is faced with, in terms of arrival rates and request properties. Each server in the cluster on its own, acts as a stand-alone web services middleware instance for the requests it receives. Execution time predictability can be successfully achieved in them with the proposed solution for stand-alone middleware presented in Chapter 3. However, given the prominent role a dispatcher plays in the distribution of requests, it can play a significant role in further ensuring predictability of execution within the cluster. However, this would require the dispatching decisions to change from being throughput-oriented to being predictability oriented.

Achieving predictability of execution in a web services cluster is important primary for two reasons. Firstly, given the common usage of such clusters on the Internet, being able to support execution deadlines, to ensure predictability of execution and to distribute requests among the cluster based on their completion deadlines will make it possible

for these clusters to be used with applications with such stringent quality requirements. Secondly, such a cluster setup will serve as a solution to reduce the amount of request rejections resulted with stand-alone web services middleware.

4.2 Problem Statement

Request dispatching techniques can be categorised in various ways [Cardellini et al., 1999, 2002; Gilly et al., 2011]. In the scope of this research we consider them categorised broadly into the following two categories.

- Request-blind dispatching schemes - Dispatcher is unaware about the incoming request and dispatching decisions is based on a pre-defined criteria.
- Request-aware dispatching schemes - Dispatcher is aware of the request and dispatching decisions are made based on some property of the request.

Request blind dispatching [Cardellini et al., 2002] schemes work mostly on the network or transport layer. They are typically used in balancing the workload among the cluster servers. By doing so, they try to achieve better overall response times and resource usage in a cluster while increasing availability and scalability. For instance, simple dispatching schemes such as Round-Robin dispatching was found to be effective in balancing the load among cluster servers [Gilly et al., 2011]. Request-aware dispatching schemes typically work on the application layer and some are designed to balance the load among cluster members. However, some of them are designed to unbalance the load within a cluster and achieve differentiated service processing among requests. For instance, many of such schemes follow some predefined scheme such as task size or customer category to differentiate service among several request classes [García et al., 2009; Pacifici et al., 2005]. A dispatcher using such a scheme may dispatch requests belonging to these classes in a pre-defined ratio among the cluster members. Such a technique will naturally result in each executor having a different workload.

While dispatching techniques make it possible to achieve better response times, scalability, availability and service differentiation in such clusters, none of the currently available techniques consider predictability based attributes such as execution deadlines or laxity of requests in their dispatching decisions. Moreover, none of the software components that govern a cluster is designed for such a purpose. Therefore, research into

such dispatching techniques and cluster architectures is of paramount importance to the success of the cluster based web service deployments.

4.3 Overview of the Solution

Our contribution through this chapter are four request dispatching algorithms that are designed to guarantee high levels of execution time predictability in web service clusters. Dispatching decisions in two of the algorithms (namely RT-RoundRobin and RT-Sequential) are done in a request-blind manner. In their functionality they try to balance the load among the cluster members and achieve the optimal scheduling and resource usage, respectively. The remaining two algorithms (RT-ClassBased and RT-LaxityBased) make dispatching decisions in a request-aware manner. RT-ClassBased considers the size of a request and assigns them to pre-designated dispatchers while RT-LaxityBased considers the laxity property of a request when requests are assigned to executors.

Apart from their specialised dispatching decisions, all four algorithms carry out the additional step of checking the schedulability of a request prior to accepting it for execution. This is achieved by incorporating the schedulability check presented in chapter 3 in the algorithms. After matching a request to a dispatcher, each algorithm considers the laxity of the request and checks with the chosen executor on whether the perceived execution deadline of the request can be met.

RT-RoundRobin is proposed as an example of how a simple dispatching algorithm can be modified to consider the deadline and laxity of a request in its functionality. Firstly, it matches a request to an executor where the executors are picked in round-robin fashion. As a second step the possibility of meeting the deadline of the target request is checked with the executor using the schedulability check. RT-ClassBased is proposed as a representation of any class based dispatching algorithm that unbalances the load in a cluster. Requests are classified into different classes based on an attribute and the mapping of these classes to executors is pre-defined. For this implementation, we consider the request size as the classifying attribute and assign each executor with a range of request sizes for execution. RT-ClassBased works by mapping a request to the corresponding executor based on its size and then checking its schedulability on that executor. RT-LaxityBased extends this process further by keeping track of the last two laxities assigned to every executor and preventing requests with similar laxities being

assigned to them consecutively. This process broadens the range of laxities further for RT-LaxityBased algorithm. Every algorithm except RT-Sequential checks the schedulability of a request only with a single executor. RT-Sequential takes this further by checking the schedulability of a request with every executor in the cluster until a proper fit is found or the list is exhausted. This allows RT-Sequential to achieve the best rates of request acceptance among all algorithms.

The rest of the chapter is organised as follows. In the next Section we discuss some of the related work in this area. Next, detailed descriptions of the four request dispatching algorithms are presented. In Section 4.6 an analytical evaluation of each algorithm is presented. It is followed by a brief Section (5.6) describing how the algorithms were implemented using actual middleware products. An empirical evaluation is presented in Section 6.8, where these algorithms are tested under various traffic conditions,their performance and predictability gain compared with other dispatching algorithms. Finally a conclusion to the chapter is provided in Section 4.9.

4.4 Related Work

Many previous attempts at achieving better response times by using a cluster setup can be found in literature. The simplest form tries to dispatch requests either equally among cluster members (i.e. using Round-Robin or Random dispatching) so that the average response times become better. Many approaches go a step forward in making the dispatching decision based on some attribute such as residual load of an executor, content requested, geographical region, popularity of a domain, etc. A popular way of such dispatching is to do it as Authoritative Dynamic Naming Service (A-DNS) level redirection [Cardellini et al., 2003; Colajanni and Yu, 2002]. Many other attempts make content-aware dispatching decisions with client or server information. For instance, requests for a particular object can be directed the same server until it reaches a particular utilisation level. By doing so, the object is loaded into a cache and served from it for subsequent requests. Similarly, requests could be classified into CPU bound or Disk Bound and served at different servers [Casalicchio et al., 2002].

Many of the previous attempts are on clusters hosting static web content and follow the premise of static web traffic taking a heavy tailed distribution. In [Mor Harchol-Balter, 2002], no prior knowledge of task sizes are assumed and requests are sent through several executors assigned with increasing quanta until the request is completed in a

non-work conserving manner. Requests are mapped to executors based on task sizes in [Ciardo et al., 2001; Harchol-Balter et al., 1999] and as a result the dispatching transforms the heavy tailed work load into that of type exponential. These work with the goal of reducing the mean waiting time, mean slowdown of tasks and not the predictability of execution. Moreover, with the assumptions they make on static web content such as the heavy-tailed nature of traffic, they seem unsuitable to be used with the highly dynamic nature of web services.

Mechanisms of achieving perceived performance levels as outlined in SLAs can also be seen in clusters hosting web services. The work of Pacifici G. et al. [Pacifici et al., 2005] uses a multi-level dispatching technique where a layer 4 switch acts as the first level dispatcher which distributes requests among several gateways in the cluster in a content-blind manner. Pre-defined SLAs classify requests in to several grades where customers pay to be in a certain grade with a probabilistic guarantee on execution times. Gateways dispatch requests among cluster servers hosting identical content and a global dispatcher keeps track of the server resources used and currently available at each server. A utility function is used by the resource manager to compute the resource consumption and calculate the number of connections from each grade a server could handle in a given window of time. This information is disseminated periodically among the gateways, which make use of them for dispatching decisions. García D. et al. [García et al., 2009] takes a similar approach where an SLA is used to specify the maximum response times for each service delivered by the provider. Each customer is guaranteed a probabilistic measure of the response times specified in the SLA. Cluster servers host identical content and a monitoring module in each server keeps track of the resource use and request execution. This information is periodically updated at a controller module which compares the information with the perceived response times on the SLA. Using the calculated statistics, the controller decides on the acceptance of a request for execution upon being queried by the load balancer. Continuation of this process leads to dynamically adjusting the request acceptance to achieve the probabilistic measures of execution times for each client. The work of Gmach D. et al. [Gmach et al., 2008] takes a different adaptive approach by using fuzzy logic to optimise parameters such as resource availability, execution times for each class of requests and performance levels perceived in the SLA. A management module uses fuzzy logic to calculate the optimal parameters for the servers where requests of certain classes will have priority over others. Similarly, Cao J. et al. [Cao et al., 2010] presents a Jini based self-configurable

service process engine which dynamically balances the load among services hosted, based on a predefined model. A heuristic technique is used to classify every request, using a tag specifying the workload it would incur on execution. The workload tags are intended for a controller module, which dynamically configures the engine based on this information using a fuzzy control algorithm. The heuristic algorithm calculates the workload incurred by a request, using a set of probability based values for each function the engine must perform to complete the request. The fuzzy control function maps this value into a generalised fuzzy number based on predefined functions that classify a request based on its resource requirement. Depending on the projected workload the control module dynamically increases or decreases the active service instances to maintain the perceived level of performance. All these approaches discussed considers service execution time as a QoS parameter and try to achieve pre-determined levels of performance. As it is a probabilistic measure, none of them can guarantee it in a consistent manner.

Similar attempts at achieving different levels of performance through service differentiation could be seen in serving simple web requests. Eggert L. et al. [Eggert and Heidemann, 1999], introduces an application level service differentiation to a web server with two service classes. Processes that serve the web requests are grouped into one service class named foreground processes and others such as cache managers that uses speculative pull and push transactions are categorised into the another as background processes. Foreground processes are comparatively more CPU bound while background processes are more network bound. The work presents three different backgrounding mechanisms that allocates processing resources between these two classes in different ratios, thereby controlling the number of processes from both classes that use the CPU. Kihl M. et al. [Kihl et al., 2008] presents an admission control mechanism for web servers which rejects requests based on load conditions at the server. Their solution uses a combination of queueing theory and control theory to ensure the CPU utilisation is preserved at a certain level. Queueing theory is used to model an Apache web server as a GI/G/1 system and the control theory is used to decide on the number of requests accepted. While these attempts succeed at introducing service differentiation or selective request execution, none of them have all the components needed to ensure consistent predictable execution times. Moreover, being application level solutions they lack support from the system level, that would ensure the required level of consistency.

4.5 Proposed Real-time Dispatching Algorithms

The proposed four dispatching algorithms are presented in this section. Apart from matching a request to an executor, all of them considers the execution deadline of the request and its laxity to ensure that the perceived deadline can be met on the selected executor. The schedulability check algorithm presented in chapter 3 (section 3.5.2) is used for this purpose. The algorithms achieve three goals in their functionality. Firstly, the requests are distributed amongst the executors to either balance (RT-RoundRobin, RT-Sequential, RT-LaxityBased) or unbalance the load in the cluster (RT-ClassBased) depending on their dispatching technique. Secondly, a request is only dispatched to the selected executor on he guarantee that its execution deadline could be met. Finally, apart from the distribution of requests among executors, the selected requests result in a wider range of laxities at each executor, allowing the requests with overlapping deadlines to be scheduled together. Following is a detailed discussion on each algorithm.

4.5.1 RT-RoundRobin

RT-RoundRobin is an example of how a simple request dispatching algorithm could be modified to consider execution deadlines and request laxities in its dispatching process. It works by matching a request to an executor in round-robin fashion and then checking the schedulability of the request on the selected executor. The additional step of request selection by the schedulability check based on its laxity results in a large range of laxities at each executor.

The functionality of RT-RoundRobin helps in achieving the deadlines of the selected requests in two ways. Firstly, the wider range of laxities created by the selection process ensures that a proper mix of larger and smaller laxities are selected, such that the execution of requests with larger laxities can be delayed or phased out to schedule requests with overlapping deadlines and smaller laxities. Moreover, the round-robin nature of the algorithm reduces the arrival rate of requests at each server (compared to the arrival rate at the dispatcher) which contributes to requests arriving further apart and less number of requests vying for the execution window of time.

Algorithm 2 details the steps in RT-RoundRobin. The round robin nature of it is maintained by keeping track of the last executor a request was assigned to (L) and assigning the new request to the next executor in the list. Upon exhausting the list of executors

after a complete round of assignments the index used to keep track of the executors (*lastExecIndx*) is reset to the beginning (Lines 2-4). Otherwise, the position of the next executor is selected by simply increasing the index (Line 5). Using *lastExecIndx*, a reference to the next executor is obtained (Line 7) and the schedulability of the new request is checked on that executor (Line 8).

Algorithm 2 RT-RoundRobin

Require: New request R, List of Executors E, Last Executor L
Ensure: R assigned to an executor or rejected

```
1.  lastExecIndx ← L.getIndex
2.  if lastExecIndx = E.size-1 then
3.      lastExecIndx = 0
4.  else
5.      lastExecIndx ← lastExecIndx + 1
6.  end if
7.  nextExec ← E.getExec(lastExecIndx)
8.  S ← IsSchedulable(R,nextExec)
9.  if S = true then
10.     L ← nextExec
11.     Assign R to nextExec
12. else
13.     Reject R
14. end if
```

If the request is schedulable, it is assigned to the executor (Lines 9-11) and a reference to the executor is kept track of as the last one to be successfully assigned a request (Line 10). A failure in the schedulability check results in the request being rejected (Line 13). Objects representing executors are kept in a data structure with constant time access when an index is used. Coupled with the schedulability check that creates a large range of laxities at each executor and deadline based scheduling, the cluster is able to achieve predictable execution times for requests accepted. Moreover, RT-RoundRobin is the simplest of the algorithms with the possible processing overhead kept to a minimum.

Complexity Analysis of RT-RoundRobin

We assume that the list of executors (E) are kept in a data structure with constant access time when an index is used. We also consider the dispatching of request R to the selected executor as an activity outside the scope of the algorithm and considers the assignment

to be done (Line 11) in constant time. Similarly, the notification to the client about the rejection of R (Line 13) is also considered as out of the scope of this algorithm and is assumed to take constant time.

Given the condition in Line 2, either outcome results an operation that executes in constant time. Similarly, the condition in Line 9 also results either way in an operation in constant time. Let n be the number of requests already assigned to the selected executor. Let $A(n)$ be the running time of Algorithm 2. Let t_1 be the total time taken to execute lines 1-6. Let t_2 be the constant time taken to retrieve the next executor using the index (*lastExecIndx*) from the list of executors. Let t_3 be the total time required to execute lines 9-14. Let t_4 be the time taken for the schedulability check on the selected executor with only a single request assigned to it already. The running time of Algorithm 2 could be defined as,

$$\begin{aligned} A(n) &= t_1 + t_2 + t_3 + n(t_4) \\ &= t_1 + t_2 + t_3 + n(t_4) \leq n(t_1 + t_2 + t_3 + t_4) \end{aligned}$$

We could conclude that the algorithm results in a worst case time complexity of $O(n)$, due to the complexity of the schedulability check. Recall that the schedulability check has a best case time complexity of $\Omega(1)$ when there are no already accepted requests at the server. Given this, we could conclude that Algorithm 2 also has a best case time complexity of $\Omega(1)$.

4.5.2 RT-ClassBased

RT-ClassBased algorithm represents all algorithms that divide requests into different classes based on a pre-defined classification and uses a static request class to dispatcher assignment. We used task size as the classification criteria for this research. The task size range was equally divided into two or more classes and each server is assigned with the execution of requests belonging to a single class. This mapping is worked out at design time of the system. The size-based classification makes RT-ClassBased an example of of a dispatching technique that favours certain classes of requests (in this case the small sized tasks) by unbalancing the load of the cluster.

RT-ClassBased works by considering the size of a task upon its arrival and obtaining the designated executor by using a simple calculation which will be described below. As a second step, it checks the schedulability of the task with the designated executor using the execution deadline and laxity of the request. The algorithm achieves three main outcomes. Firstly, by conducting size-based dispatching it achieves better waiting times and slowdown for smaller sized tasks. The task based segregation between the executors ensure that the small requests are not made to wait for the completion of large sized tasks. Secondly, the use of schedulability analysis ensures that the designated executor could indeed achieve the perceived deadline of requests. Finally, the range of laxities at each executor resulted by the schedulability check maximises the chances of executing requests with overlapping deadlines, together.

Algorithm 3 contains the steps of RT-ClassBased. Recall that we equally divide the task size range amongst the executors. Firstly, the limit L that decides the size of the range is calculated by dividing the total of the smallest and largest task sizes, by the number of classes or executors in the cluster (Line 1). Secondly, the task size is obtained from the new request R (Line 2) and the request class is obtained by an integer division of the size of R by limit L (Line 3). The resulting integer value corresponds to the index of the respective executor that size is assigned to. Therefore the result is used to lookup the executor for the list of executors (Line 4).

Algorithm 3 RT-ClassBased

Require: New request R, List of Executors E, Number of Classes N, Smallest Size SM, Largest Size LG
Ensure: R assigned to an executor or rejected
1. L $\leftarrow \left(\frac{SM + LG}{N}\right)$
2. SZ $\leftarrow$ R.getSize
3. C $\leftarrow \left(\frac{SZ}{L}\right)$
4. nextExec $\leftarrow$ E.getExec(C)
5. S $\leftarrow$ IsSchedulable(R,nextExec)
6. **if** S = true **then**
7. Assign R to nextExec
8. **else**
9. Reject R
10. **end if**

Thereafter, the request is directly checked for schedulability with the selected executor (Line 5) and assigned to it on a successful outcome (Line 7). If the schedulability check

fails, the request is rejected (Line 9). Line 2 shows the size of the request being retrieved from itself for brevity. As there is no knowledge of a request prior to its arrival, the size of the request has to be inferred from information at hand. Profiled execution times or execution time history can be used for this purpose.

Complexity Analysis of RT-ClassBased

We make the assumption that the list of executors are kept in a data structure with constant access time when accessed using an index. We also assume that the dispatching of request R to the selected executor as an activity outside the scope of the algorithm and considers the assignment to be done (Line 7) in constant time. Similarly, the rejection of R (Line 9) is also considered as out of the scope of this algorithm and assumed to take constant time. Furthermore, we assume that the method of obtaining the task size of the new request will also result in constant time access.

Each statement leading up to line 5 takes constant execution time. Similarly, either outcome of the conditional statement in line 6, also result in constant execution time. Recall from the complexity analysis presented in Chapter 3, that the schedulability check has a worst case linear time complexity and a best case constant time complexity.

Let n be the number of requests already assigned to the selected executor. Let $B(n)$ be the running time of Algorithm 3. Let t_1 be the total time taken to execute lines 1-4. Let t_2 be the total time required to execute lines 6-10. Let t_3 be the time taken for the schedulability check on the selected executor with only a single request assigned to it already. The running time of Algorithm 3 could be defined as,

$$\begin{aligned} B(n) &= t_1 + t_2 + n(t_3) \\ &= t_1 + t_2 + n(t_3) \leq n(t_1 + t_2 + t_3) \end{aligned}$$

Therefore we could conclude that the worst case time complexity of Algorithm 3 is linear ($O(n)$) due to the worst case time complexity of the schedulability check. Furthermore, with the best case time complexity of the schedulability check being constant access time, Algorithm 3 also has a best case complexity of $\Omega(1)$.

4.5.3 RT-LaxityBased

RT-LaxityBased algorithm is a request-aware dispatching scheme where requests are mapped to executors based on the laxity property of a request. The other three proposed algorithms only make use of the laxity in the schedulability check when the possibility of achieving the execution deadline of a request is checked with the selected executor. RT-LaxityBased uses the laxity property furthermore in its selection of an executor for a request.

One outcome of the schedulability check is the range of laxities it results in at each executor. Having such a mix of laxities paves the way for more requests with overlapping lifespans to be scheduled together. This is achieved by delaying or phasing out the execution of requests with higher laxity values, making way for requests with lower laxities to be scheduled within their time frame. RT-LaxityBased aims to better this process by keeping track of the laxities assigned to each executor. It works by storing the last two laxity values assigned to each executor and preventing requests with the same laxity values being assigned to the same executor consecutively. Moreover, it keeps track of the last executor to have a request assigned to and considers a different executor for the next request. This process leads to an increased range of laxities at an executor and enables more requests to be scheduled together.

Algorithm 4, describes the steps in RT-LaxityBased. Upon the arrival of a request, its laxity is calculated (Line 1). It is checked to ensure not to be one of the last two laxities assigned to the executor (Line 3-4). If the laxity is not one of the immediate previous values assigned, the request is checked for schedulability with the last executor (Line 5). If the request could be scheduled successfully (Line 6) it is assigned to the executor (Line 8) after the laxity value is recorded as one of the two values to be successfully assigned (Line 7). If the schedulability check fails, the request is rejected (Lines 9-11).

In the case of the calculated laxity being in the last two laxities assigned to the last executor, next executor in the list is considered (Lines 13-14). The last two laxities assigned to that particular executor is obtained and the calculated laxity is checked against them (Lines 16-17). If it is found to be one of them as well, the process continues on to consider subsequent executors in the list until a match is found (Line 13). If the laxity is not one of them, the schedulability check is done on the selected executor (Line 18-19) and the request is either assigned to it or rejected based on the result (Lines 19-24). The first time a request is scheduled through the algorithm, there is no last executor

Algorithm 4 RT-LaxityBased

Require: New request R, List of Executors E, Laxity Map LM, Last Executor L
Ensure: R assigned to an endpoint or rejected

```
1.  Laxity ← (R.getDeadline / R.getExecutionTime)
2.  if lastExec is not ∅ then
3.     LL ← lastExec.LastLaxities
4.     if Laxity is not in LL then
5.        S ← IsSchedulable(R,lastExec)
6.        if S = true then
7.           lastExec.setLastLaxities(Laxity)
8.           Assign R to lastExec
9.        else
10.          Reject R
11.       end if
12.    else
13.       while E.hasMore() and R is not assigned and R is not rejected do
14.          nextExec ← E.getNextExec
15.          if nextExec is not lastExec then
16.             LL ← nextExec.LastLaxities
17.             if Laxity not in LL then
18.                S ← IsSchedulable(R,nextExec)
19.                if S = true then
20.                   nextEx.setLastLaxities(Lax)
21.                   lastExec ← nextExec
22.                   Assign R to nextExec
23.                else
24.                   Reject R
25.                end if
26.             end if
27.          end if
28.       end while
29.    end if
30. else
31.    nextExec ← E.getfirstExec
32.    S ← IsSchedulable(R,nextExec)
33.    if S = true then
34.       nextExec.setLastLaxities(Laxity)
35.       lastExec ← nextExec
36.       Assign R to nextExec
37.    else
38.       Reject R
39.    end if
40. end if
```

information available. In such a scenario the request is checked for schedulability with the first executor in the list (Lines 30-40). The calculated laxity is recorded as the first laxity to be assigned to that executor (Line 34).

Complexity Analysis of RT-LaxityBased

We assume that executor information and details of last laxities assigned to executors are kept in data structures with linear and constant access time complexities respectively. We also assume that dispatching R to the selected executor and confirming the rejection of R to the client is outside of the scope of this algorithm and consider the execution time taken of those steps to be constant.

The best case execution for Algorithm 4 would be on the arrival of the first request at the system. In which case the condition on line 2 evaluates to be *false* and the statements 31 to 40 gets executed. As line 32 contains a schedulability check, of which the execution time complexity is known to be $O(n)$ and $\Omega(1)$. The rest of the statements within lines 31 to 40 have constant time execution. Therefore the worst case time complexity of lines 31 to 40 can be concluded as $O(n)$. If the condition on line 2 evaluates to *true*, the execution can again take two paths at line 4. If it evaluates to *true*, Line 5 contains a schedulability check and rest of the statements in lines 6 - 11 result in constant time. If the condition on line 4 evaluates to *false*, line 13 has a *while* loop that iterates at most equal to the number of executors in the cluster. Note that although there are two conditions at lines 15 and 17, either one of them evaluating to *false* will result in a loop iteration or a loop exit (when the list of executors has been exhausted). If both conditions evaluate to be *true*, statements on lines 18-25 will be executed. In which case every statement except for the schedulability check on line 18 would have constant time execution. Note that although the schedulability check on line 18 is within the *while* loop, it will only be executed once as the condition on line 17 evaluating to *true* will result in the termination of the *while* loop (due to lines 22 and 24).

Let m be the number of executors in the cluster. Let n be the number of already assigned requests at the selected executor. Let $C(n)$ be the running time of Algorithm 4. Let t_1 be the execution time of the laxity calculation on line 1 and the condition on line 2. Let t_2 be the execution time of lines 3-4. Let t_3 be the execution time taken for statements 6-11. Let t_4 be the time taken for execution of statements 13-17. Let t_5 be the execution time of statements 19-25. Let t_6 be the execution time for statements 31 and 33-40. Let

t_s be the time taken for the schedulability check on the selected executor with only a single request assigned to it already. The running time of Algorithm 4 can be defined as,

$$\begin{aligned} C(n) &= t_1 + t_6 + n(t_s) \mid t_1 + t_2 + n(t_s) + t_3 \mid t_1 + t_2 + m(t_4) \mid t_1 + t_2 + n(t_s) + t_5 \\ &= t_1 + t_6 + n(t_s) \leq n(t_1 + t_6 + t_s) \mid t_1 + t_2 + n(t_s) + t_3 \leq n(t_1 + t_2 + t_s + t_3) \mid \\ &\quad t_1 + t_2 + m(t_4) \leq m(t_1 + t_2 + t_4) \mid t_1 + t_2 + n(t_s) + t_5 \leq n(t_1 + t_2 + t_s + t_5) \end{aligned}$$

Given the definition above, $C(n)$ can be considered to be linear. However, it is fair to assume that the number of executors (m) can be considered $\forall m, m < n$. Therefore, the worst case time complexity of Algorithm 4 is $O(n)$ due to the schedulability check. Furthermore, its best case time complexity can be conclude as $\Omega(1)$.

4.5.4 RT-Sequential

All three of the previous algorithms had the common feature of checking the schedulability of a request at most with one executor in the cluster. RT-Sequential algorithm on the other hand, checks the possibility of achieving the execution deadline of a request with multiple executors. In turn it tries to make best possible use of the server resources available on the cluster. If the schedulability check for a request fails with one executor, RT-Sequential continues to exhaustively check its schedulability with the rest of the executors in the cluster until it is schedulable on one of them or the list exhausted. Although this is somewhat a request-blind dispatching scheme, like RT-LaxityBased this achieves a larger range of laxity at an executor due to fitting a request ultimately to the best executor. However,it does this with the additional cost of multiple schedulability checks per request. The other algorithms keeps it to a minimum to prevent this cost being too significant, as the lifetime of a request starts from the moment it enters the system.

Algorithm 5 details the steps in RT-Sequential. To prevent RT-Sequential always starting with the same executor, the successful executor from the last run is kept track of and is considered first (Lines 1,11,22). Requests are repeatedly checked for schedulability on it until the check fails (Lines 1-4), in which case another executor is considered

(Lines 6-16). This process continues on until one of two outcomes. A request may ultimately be found to be schedulable in a subsequent executor (Lines 20-23). The other being the entire list of executors being exhausted with the inability to find an executor having already accepted requests with complementing laxities. In which case the request is ultimately rejected (Lines 27-29).

Algorithm 5 RT-Sequential

Require: New request R, List of Executors E, Last executor
Ensure: R assigned to an executor or rejected

```
1.  if lastExec is not ∅ then
2.     S ← IsSchedulable(R,lastExec)
3.     if S = true then
4.        Assign R to lastExec
5.     else
6.        while E.hasMore() AND R not assigned do
7.           nextExec ← E.getNextExec
8.           if nextExec is not lastExec then
9.              S ← IsSchedulable(R,nextExec)
10.             if S = true then
11.                lastExec ← nextExec
12.                Assign R to nextExec
13.             end if
14.          end if
15.       end while
16.    end if
17. else
18.    while E.hasMore() AND R not assigned do
19.       nextExec ← E.getNextExec
20.       S ← IsSchedulable(R,nextExec)
21.       if S = true then
22.          lastExec ← nextExec
23.          Assign R to nextExec
24.       end if
25.    end while
26. end if
27. if R is not assigned then
28.    Reject R
29. end if
```

Complexity Analysis of RT-Sequential

We assume that executor information is kept in a data structure with linear access time when accessed sequentially. As in the previous algorithms we assume the actual dispatching of the request and any request rejections to be out of scope of the algorithm and considers the corresponding statements to be executing in constant time.

The execution of Algorithm 5 can take one of three paths. The first of the three is when the condition on line 1 evaluates to be *true* and the request is schedulable on the last executor considered. The second execution path is when the request is not able to meet its deadline on the last executor, therefore the algorithm checks its schedulability with other executors in the cluster until a match is found or the list is exhausted (Lines 6-15). The third and final execution path is when condition on line 1 evaluates to be *false* where the request is checked for schedulability with the next available executor on the list. Herein, the same process of exhausting checking continues similar to the second execution path until a match is found or the list is exhausted. A noteworthy observation is that a schedulability check happens in every execution path and all other statements execute in constant time. The best case scenario for this algorithm is when there no requests already assigned at the selected executor. In which case the schedulability check is only done once and in its best case execution in constant time.

Let m be the number of executors in the cluster. Let n be the maximum number of already assigned requests found in any of the executor. Let $D(n)$ be the running time of the algorithm. Let t_s be the execution time taken for the schedulability check with only a single request already assigned to the selected executor. Let t_1 be the maximum time taken for execution of statements 1 and 27-29. Let t_2 be the maximum time taken for the execution of statements 3-5. Let t_3 be the worst case time taken to execute statements 6-8 and 10-15. Let t_4 be the worst case execution time of statements 18-19 and 21-26. The running time of Algorithm 5 can be defined as,

$$
\begin{aligned}
D(n) &= t_1 + n(t_s) + t_2 \mid t_1 + m(t_3) + m(n(t_s)) \mid t_1 + m(t_4) + m(n(t_s)) \\
&= t_1 + n(t_s) + t_2 \leq n(t_1 + t_s + t_2) \mid t_1 + m(t_3) + m(n(t_s)) \leq mn(t_1 + t_3 + t_s) \mid \\
&\quad t_1 + m(t_4) + m(n(t_s)) \leq mn(t_1 + t_4 + t_s)
\end{aligned}
$$

With this definition, it can be concluded that the worst case time complexity of Algorithm 5 is linear and in the order of $O(mn)$ due to the schedulability check being done multiple times. Furthermore, the best case execution scenario for the algorithm is when the schedulability check is done only ones and there are no already accepted requests assigned to the selected executor, which gives the algorithm a best case time complexity of $\Omega(1)$ equal to that of the schedulability check.

Except for RT-Sequential algorithm, the other three algorithms checks the schedulability of a request only with one executor. If the schedulability check fails, the request is rejected outright. Due to the worst case time complexity of the schedulability check, it was decided to keep the number of checks per request to a minimum where possible. This would prevent the additional time required by the schedulability check becoming and overhead when a request is scheduled. Due to the algorithms being designed to distribute the requests among cluster members, the amount of requests directed to an executor is lower compared to a single host scenario. The algorithms presented in this section have been modelled to abstract the configuration of the executors in terms of number of processors or cores available.

4.6 Analytical Evaluation of the Dispatching Algorithms

The objective of this section is to provide an analytical evaluation of the functionality of the proposed algorithms. We use a created sample data set for this purpose for each algorithm separately. Note that an analytical evaluation of the schedulability analysis, the associated model and the algorithm used by these algorithms were presented in Section 3.6 of Chapter 3, Therefore, analytical evaluation of laxity based schedulability check and deadline based scheduling is not repeated in this section. However, where appropriate (for instance in RT-LaxityBased) we illustrate how they work in order to gain more clarity in the evaluation process. Note that in the evaluation of RT-RoundRobin, RT-ClassBased and RT-LaxityBased we consider the schedulability of the sample requests to be a success on the selected executors, for brevity.

Analytical Evaluation of RT-RoundRobin

To evaluate RT-RoundRobin we consider five request arrivals and consider the cluster to have three executors. Table 4.0b represents the data structure that stores the list of

executor where each executor instance is identified by the index, providing constant time access to their instances. The *lastExecIndex* keeps track of the last executor a request was assigned to and is responsible for preserving the round-robin nature of the algorithm.

	lastExecIndex
Request 1	**0**
Request 2	**1**
Request 3	**2**
Request 4	**0**
Request 5	**1**

(a) Value of lastExecIndex

Index	Executor Instance
0	Executor 1
1	Executor 2
2	Executor 3

(b) List of Executors

Table 4.1: Overview of RT-RoundRobin Properties

lastExecIndex starts off with 0 and as a result first request is assigned to Executor 1 which is at position 0 of the list. On the arrival of the second request the *lastExecIndex* is increased by one and Request 2 is assigned to Executor 2 which is at position 1 of the list. Request 3 arriving next is assigned to Executor 3 which is at position 2 of the list, following the same process. At the arrival of Request 4 the condition on line 3 holds to be *true* and as a result, the *lastExecIndex* is reset to 0 demonstrating the round-robin nature of the dispatching. Therefore, Request 4 is assigned to Executor 1 at position 0. The process continues on from there onwards as Request 5 is assigned to Executor 2 at position 1.

Note that the additional step of schedulability analysis was omitted in this analysis due to aforementioned reasons. While we considered all requests to be schedulable on the assigned executor, a failure of the schedulability check would result in the request being rejected and the subsequent request being assigned and checked for schedulability with the same executor.

Analytical Evaluation of RT-ClassBased

We evaluate RT-ClassBased with the attributes listed in Table 4.1a. We consider a cluster setup of three executors and divide the task size range equally between them. This is done by calculating a limiting value which is also used in Algorithm 3 for its cal-

culations. The task size range assigned to each executor can be seen in Table 4.1b. Moreover, it also contains the Executor instances identified by the index used in the data structure. We use five requests for this evaluation and their properties are listed in Table 4.1c.

Property	Value
Smallest Task Size	1 ms
Largest Task Size	3000 ms
Limiting Value	$\frac{1+3000}{3} = 1000$

(a) Size-based Attributes

Index	Executor Instance	Start Size	End Size
0	Executor 1	1 ms	1000 ms
1	Executor 2	1001 ms	2000 ms
2	Executor 3	2001 ms	3000 ms

(b) List of Executors

	Task Size	Calc. of Index on Arrival
Req. 1	682 ms	$\frac{682}{1000} = 0$
Req. 2	2300 ms	$\frac{2300}{1000} = 2$
Req. 3	850 ms	$\frac{850}{1000} = 0$
Req. 4	1780ms	$\frac{1780}{1000} = 1$
Req. 5	2580ms	$\frac{2580}{1000} = 2$

(c) Request Properties

Table 4.2: Overview of RT-ClassBased Properties

At the arrival of each request, the algorithm considers the task size of the request and calculates the designated executor by conducting an integer division of the task size by the limiting value. The resulting value corresponds to the index of the Executor instance. For instance, Request 1 is 682 ms in its size and therefore, will be assigned to Executor 1 identified by index 0. Similarly, Request 2 has the size of 2300 ms and therefore will be assigned to Executor 3 identified by index 2. Request 3 which falls into the range of 1 - 1000 ms gets ends up with an index of 0, therefore being properly assigned to Executor 1. This process continues on for every executor and as illustrated the integer division method can easily identify the proper executor by the use of the task size.

Herein, the schedulability check is conducted as the next step after a request is matched to an executor. A failure in it will result in the request being rejected.

Analytical Evaluation of RT-LaxityBased

RT-LaxityBased is evaluated using the requests listed in Table 4.2a. Table 4.2b contains the list of executors identified by the index of the data structure. The last two laxities assigned to each executor is also kept track of. Table 4.2c lists out the request arrivals and the value of *lastExec* and *nextExec* at the arrival and at the end of the dispatching operation respectively.

	Size	Deadline	Laxity
Req. 1	100 ms	1000 ms	10
Req. 2	200 ms	1000 ms	5
Req. 3	1500 ms	7500 ms	5
Req. 4	2700 ms	7100 ms	3
Req. 5	500 ms	5000 ms	10
Req. 6	3000 ms	9000 ms	3

(a) Request Properties

Index	Executor Instance	Last 2 Laxities	
0	Executor 1	10	5
1	Executor 2	10 $\leftarrow$ ~~5~~	3
2	Executor 3	3	

(b) List of Executors

Arrival	*lastExec*	*nextExec*
1	-	0
2	0	0
3	0	1
4	1	1
5	1	1
6	1	2

(c) Task Arrivals

Table 4.3: Overview of RT-LaxityBased Properties

Table 4.2a contains the size, deadline and the laxity of each request. Since Request 1 arrives first at the system it is straightaway assigned to Executor 1 and its laxity is kept track of against the executor. The index of Executor 1 which is in *nextExec* is copied to *lastExec* at the end of the assignment. At the arrival of Request 2, *lastExec* has that value and since Request 2 has a different laxity it is assigned to the same executor and the laxity value recorded. Request 3 has the same laxity as Request 2, as a result the algorithm assigns it to the next executor in the list which is Executor 2. This changes the values of *nextReq* and *lastExec* for the next request arrival. Request 4 having is also assigned to Executor 2 for having a different laxity. Request 5 has the laxity of 10 which is not in the last two laxities assigned to Executor 2 therefore, it is assigned to the same executor and the oldest laxity out of the two (which is 5) is replaced by 10. Subsequently, Request 6 with a laxity value of 3 is assigned to Executor 3 as the laxity is one of the last two values assigned to Executor 2. This process continues similarly

for all requests.

Similar to the other algorithms the schedulability of a request is checked as the second step once it is matched to an executor. If it cannot be scheduled in the selected executor, it is rejected.

Analytical Evaluation of RT-Sequential

RT-Sequential is evaluated using the requests in Table 4.3a which contains their sizes, deadlines and arrival times at the system (in elapsed time). The *lastExec* and *nextExec* attributes for each at each arrival is displayed in Table 4.3c. *lastExec* gets updated at the end of a task assignment therefore its effect is on the next arrival, whereas *nextExec* is used for the current task being dispatched.

	Size	Deadline	Arrival Time
Req. 1	6s	10s	0s
Req. 2	3s	7s	1s
Req. 3	3s	7s	2s
Req. 4	8s	10s	3s
Req. 5	3s	8s	4s
Req. 6	5s	8s	6s

(a) Request Properties

Index	Executor Instance
0	Executor 1
1	Executor 2
2	Executor 3

(b) List of Executors

Arrival	*lastExec*	*nextExec*
1	-	-
2	0	-
3	0	1
4	1	1
5	1	2
6	2	2

(c) Task Arrivals

Table 4.4: Overview of RT-Sequential Properties

The complete scenario under evaluation is also illustrated using Figure 4.1. On its arrival, Request 1 is directly assigned to the first executor in the list as it is the first task to arrive at the system. On the arrival of Request 2, it is checked for schedulability with Executor 1 and Request 1 is able to phase out its execution without missing its deadline to let successfully schedule Request 2, which has an overlapping and earlier deadline. Request 3 that arrives 3s into the arrival of the first request also has an overlapping deadline. However, the schedulability check fails on it as accepting it will result in Request 1 missing its deadline. Therefore, the algorithm checks its schedulability with

Executor 2, which is the next in the list. As Executor 2 has no tasks, Request 3 is scheduled on it successfully.

Request 4 has a later deadline than Request 3, however, it can be successfully scheduled on Executor 2 to meet it. A second later, Request 5 arrives at the system and is checked for schedulability with Executor 2. The check results in a failure as accepting and it is then checked for schedulability with a currently free Executor 3, and successfully dispatched to it. Request 6 that arrives at the 2s later, is also be successfully scheduled on Executor 3.

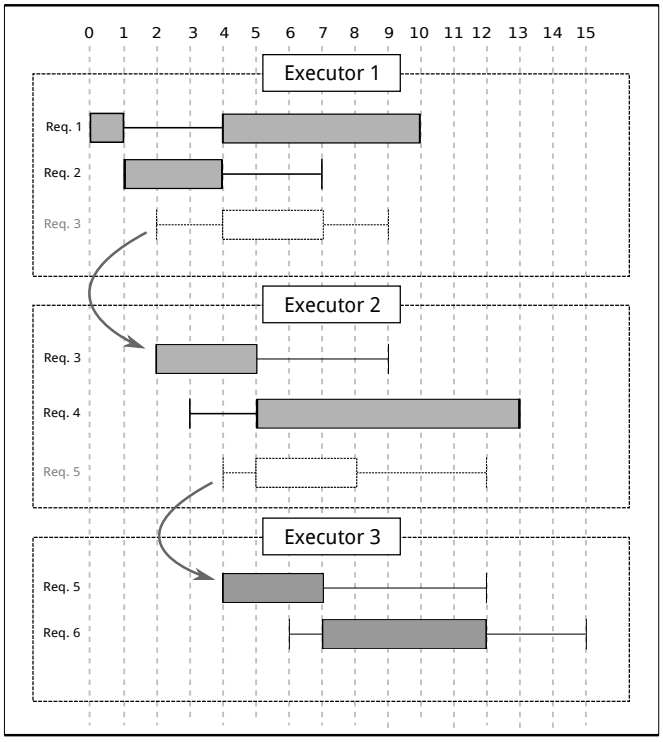

Figure 4.1: Analytical Evaluation - RT-Sequential

4.7 Implementation

The predictability gain achieved by the proposed algorithms were empirically evaluated by implementing them in a cluster based web services setup. The implementation contains two main aspects. The proposed algorithms are implemented in the dispatcher, while each executor must also ensure execution deadlines are honoured and predictability of execution is achieved.

Figure 4.2 illustrates the different components in the implementation. In chapter 3, we presented techniques for achieving predictability of execution on stand-alone web services middleware. The proposed techniques were implemented in Apache Axis2 making RT-Axis2. Detailed information about this implementation is presented in chapter 5.

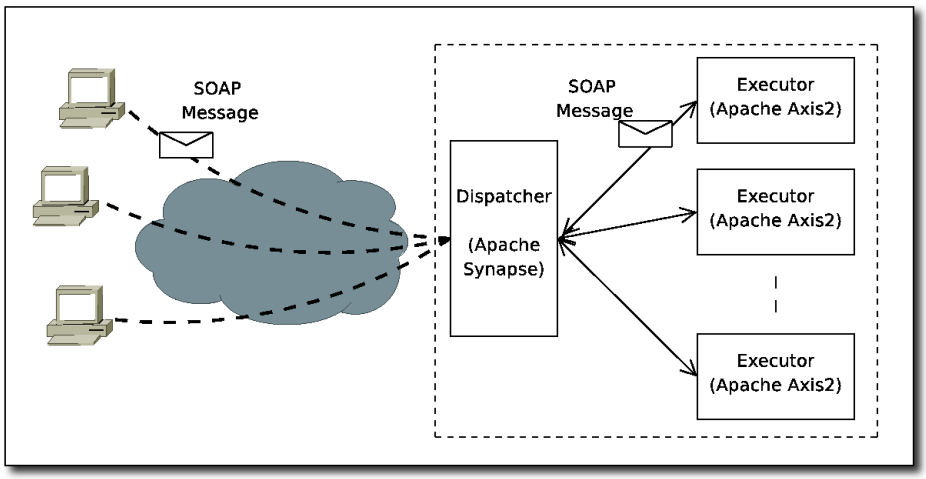

Figure 4.2: Overview of the Implementation

In implementing the cluster, we use a modified version of RT-Axis2 instances for the executors. The modification made is to free it from conducting the schedulability check and task it only with request scheduling and execution based on execution deadlines. The schedulability check is made part of the dispatchers functionality. Another open source product in Apache Synapse [Apache Software Foundation, 2008] which is a lightweight ESB implementation is enhanced to act as the dispatcher. Synapse by default has simple request dispatching capabilities. These capabilities are enhanced to implement the four proposed dispatching algorithms.

With dispatching decisions being done considering execution deadlines and request laxity, the following implementation level enhancements were made in Synapse to ensure predictability of execution is supported throughout its functionality. The execution deadline is conveyed to the dispatcher using SOAP headers and it is fetched and made available to the algorithms by modifying internal data structures of Synapse. The default *best-effort* nature of Synapse was replaced with a priority mode that can be used by an introduced real-time scheduler component to control the execution of worker

threads by the change of their priorities. The default thread-pools in Synapse were replaced with custom made real-time thread pools to manage the execution of the requests. The introduced real-time scheduler component uses multiple lanes of execution within Synapse to control the number of concurrent executions. This allows the scheduler to guarantee predictability while achieving an accepted level of throughput when processing requests. The implementation of the proposed dispatching algorithms were done using the sequence and endpoint extension framework built into Synapse. Finally, the functionality of the enhanced Synapse version (RT-Synapse) is supported by development platforms and operating systems with real-time features. A detailed discussion on the implementation aspects of the solution, software engineering techniques, designs patterns, tools used and challenges faced is presented in chapter 5.

4.8 Empirical Evaluation of the Dispatching Algorithms

4.8.1 Experimental Setup

The real-life implementation of the system using RT-Synapse and RT-Axis2 is evaluated to measure the predictability gain by the enhancements made. The implementation was hosted in a production level setup that represents a real-world deployment. Figure 4.3 illustrates the hardware and software setup used for the test environment.

The envisioned solution relies on precision of time and the implementation devised requires support for predictability at the development platform and operating system level. Thus, in setting up the web server cluster, the dispatcher was hosted on a server with a hardware configuration of 2 Intel Core 2 Duo processors running at 3.4 GHz with 4 Gigabytes of RAM. As the real-time operating system Solaris 10 update 08/05 was used with Apache web server as additional software installed. As the implementation was done in Java, Sun Java Real-Time Specification version 2.1 was installed as the platform for the real-time aware Apache Synapse version to run on. Each executor had similar hardware and software configuration. The executors were installed with a modified version of RT-Axis2 (relieved of conducting the schedulability check and tasked only with deadline based scheduling) with the web service used for the experiments deployed. The request generation was done using 5 client machines, each with an AMD Duron Processors running at 1.7 GHz speed with 1 Gigabyte of RAM. For precision of time requirements these were installed with Ubuntu Linux 8.04 running Linux real-

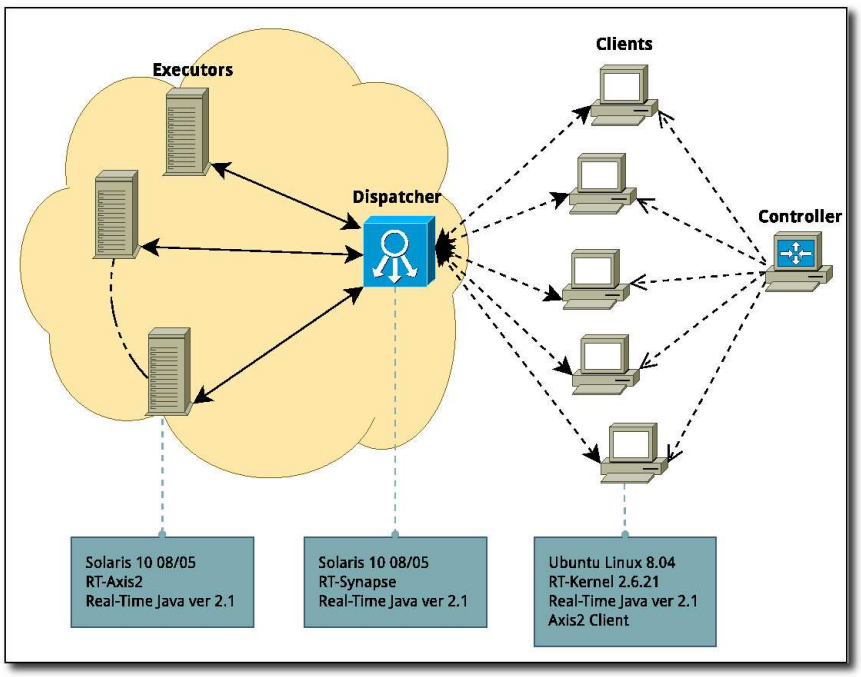

Figure 4.3: RT-Synapse Experiment Setup

time kernel version 2.6.21. The software configuration consisted of Sun Real-time Java Specification 2.1 and Apache Axis2 client libraries. A controller machine to manage the experiments, was used with a similar hardware and software configuration. The request generation software was developed using Java real-time version in order to ensure accuracy in the request inter-arrival times. Similar to the stand-alone scenario, RT-Axis2 instances were configured to have 3 lanes of execution with 30 worker threads in each lane. RT-Synapse was also configured to have 3 execution lanes with 30 threads per lane.

The web service cluster was exposed to request streams with various task size, arrival rate and deadline combinations. A particular configuration is setup at the controller and the experiment started. The controller decides on the size of a request, inter-arrival times, deadlines of a request and delegates creating the specific request to a designated client by communicating the necessary parameters. Each client is assigned a range of task sizes and requests for those sizes will only be done by the specific client. This process continues on until the specified total number of requests is reached.

As there is no evidence on a widely accepted data-set that includes web service requests with specified execution deadlines, evaluating our solution becomes a challenging task.

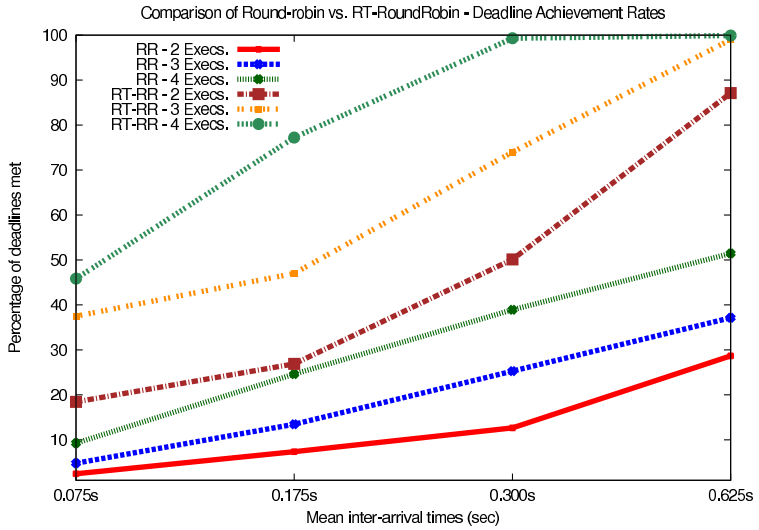

Figure 4.4: RoundRobin vs. RT-RoundRobin Deadline Achievement Rates

Therefore, as in the previous chapter, the implementation is tested for different request streams with varying properties. Generation of requests is done by using a custom made request generator where task sizes, arrival rates and deadlines are all uniformly distributed.

Evaluating the cluster based setup, the performance of the enhanced cluster is compared with a unmodified cluster for the performance of all four dispatching algorithms. The unmodified cluster consists of regular Synapse as the dispatcher which would use different algorithms for each test scenario and regular Axis2 as executors hosting the web services. The unmodified cluster dispatches and executes requests in a *best-effort* manner. The evaluation is done for various arrival rates, gradually increasing the number of executors in a cluster starting from 2 up to a maximum of 4.

4.8.2 Round-Robin Dispatching

The round-robin dispatching scenario is a fair evaluator for the predictability in execution, that could be achieved by enhancements made to a cluster based middleware setup. We compare the performance of RT-RobinRobin and simple round-robin dispatching using the same cluster based setup while increasing the request arrival rates and the cluster configurations. With the latter, there is no conditional acceptance of requests and execution happens in a *best-effort* manner. RT-RoundRobin uses the schedulability

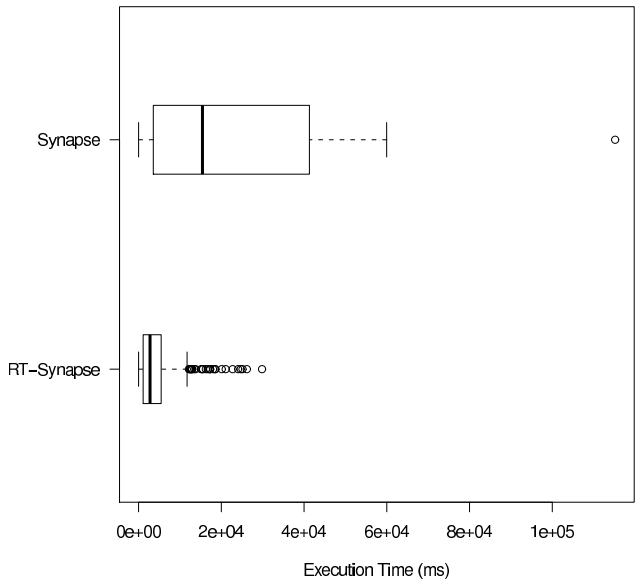

Figure 4.5: RoundRobin vs. RT-RoundRobin Execution Time Ranges

check to select requests for execution based on their laxity and selected requests get executed in the order of their deadlines. Table 4.5 summarises the results for the round-robin runs and Figure 4.4 summarises the results graphically (Note that the *x-axis* of the graph contains the mean value for the respective inter-arrival time period mentioned in Table 4.5). Due to the unconditional acceptance of requests, simple round-robin results in higher request acceptance rates. The rejection of requests when using simple round-robin dispatching, was caused by overloaded conditions resulted in the cluster. In such circumstances, the middleware and becomes unresponsive to requests and requests time out at the dispatcher as well as at the client after retrying transmissions.

While RT-RoundRobin results in lower acceptance percentages comparatively, it clearly outperforms simple round-robin in the resultant percentage of deadlines met. The best performance simple round-robin could achieve is 51.5% of the deadlines with almost all requests being accepted for execution, when 4 executors were used in the cluster. However, RT-RoundRobin, consistently achieve more than 90% of the deadlines in all the runs conducted, while maintaining decent acceptance rates. Although a higher number of requests are accepted for execution with simple round-robin scheduling, the executors get overloaded as a result of *best-effort* execution of requests. The overloading leads to the system being stalled and the overall execution of requests being delayed while other requests ready for execution are made to wait at the dispatcher. Figure 4.6

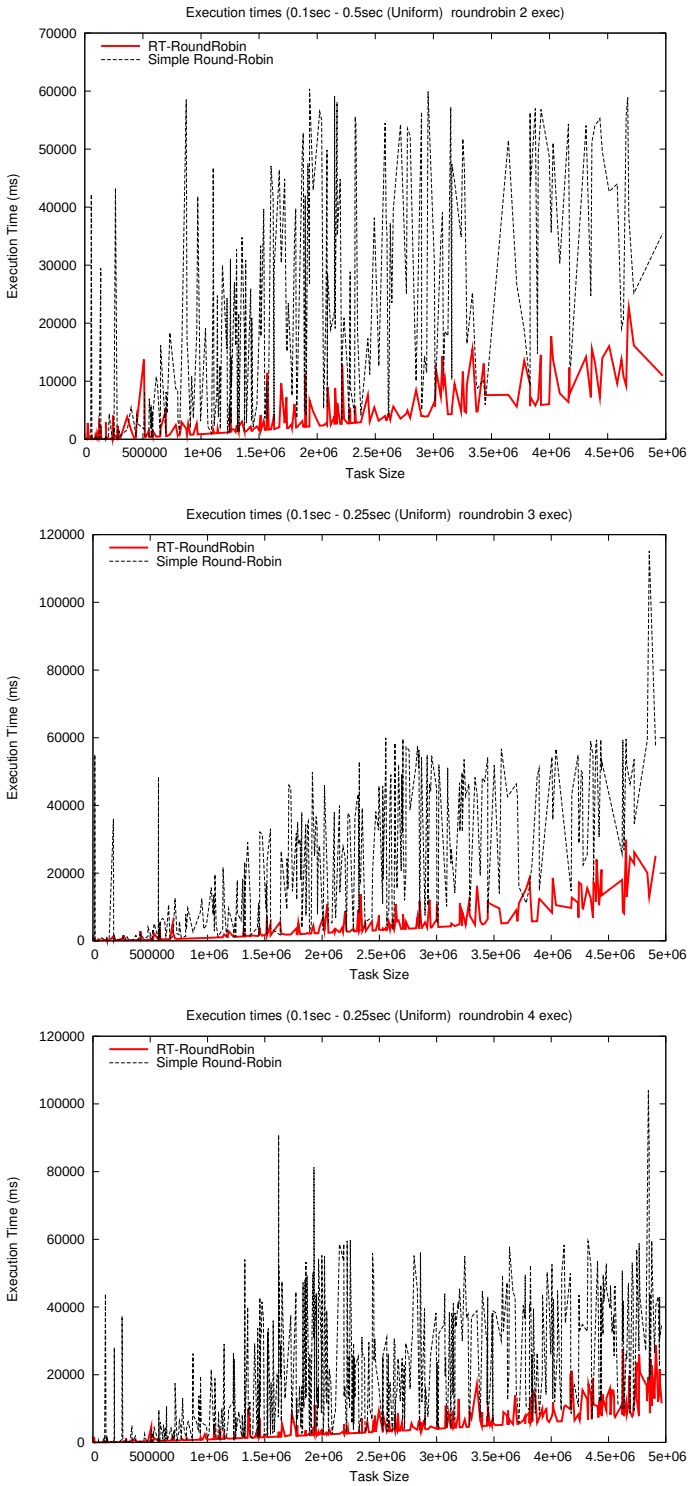

Figure 4.6: RoundRobin vs. RT-RoundRobin Resultant Execution Times

	Round Robin (Non real-time)					
	2 Executors		3 Executors		4 Executors	
Inter arrival times (sec)	% Accept.	% Dead. Met	% Accept.	% Dead. Met	% Accept.	% Dead. Met
0.25 - 1	99.5	28.8	99.8	37.2	99.9	51.5
0.1 - 0.5	62.3	20.3	89.0	28.4	98.0	39.7
0.1 - 0.25	49.0	15.0	67.3	20.0	74.1	33.2
0.05 - 1	38.8	6.3	52.6	9.1	68.0	13.6

	RT-RoundRobin					
	2 Executors		3 Executors		4 Executors	
Inter arrival times (sec)	% Accept.	% Dead. Met	% Accept.	% Dead. Met	% Accept.	% Dead. Met
0.25 - 1	88.0	99.0	99.0	100	99.9	100
0.1 - 0.5	52.0	96.4	74.0	99.0	99.4	99.9
0.1 - 0.25	28.0	96.0	47.0	97.6	78.0	99.0
0.05 - 1	20.5	90.0	37.5	95.0	46.3	99.0

Table 4.5: Performance Comparison of Round Robin vs. RT-RoundRobin

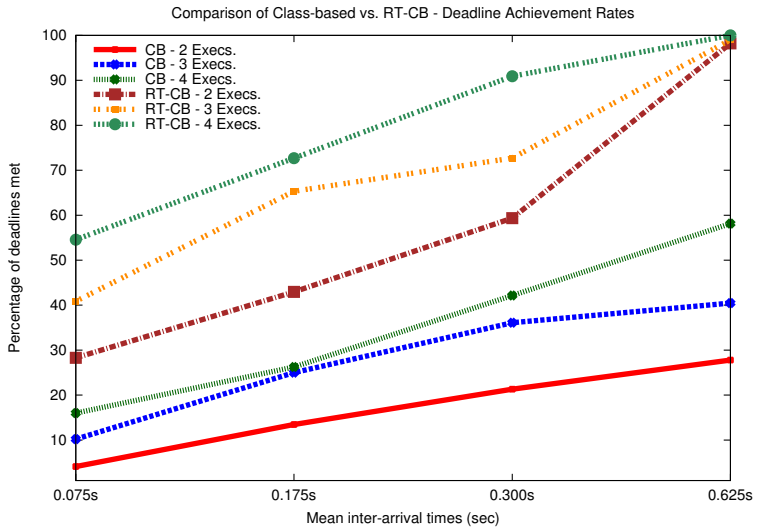

Figure 4.7: Class Based vs. RT-ClassBased Deadline Achievement Rates

contains 3 plots of resulting overall execution times by the two systems in three different cluster configurations and traffic conditions. It is clearly visible that the *best-effort* processing results in longer execution times often surpassing the deadline requirement of requests in the simple round-robin scenarios.

4.8.3 Class-Based Dispatching

Many of the request-aware dispatching schemes map requests to servers based on a predefined conditions using some property of the request. As such, requests are divided

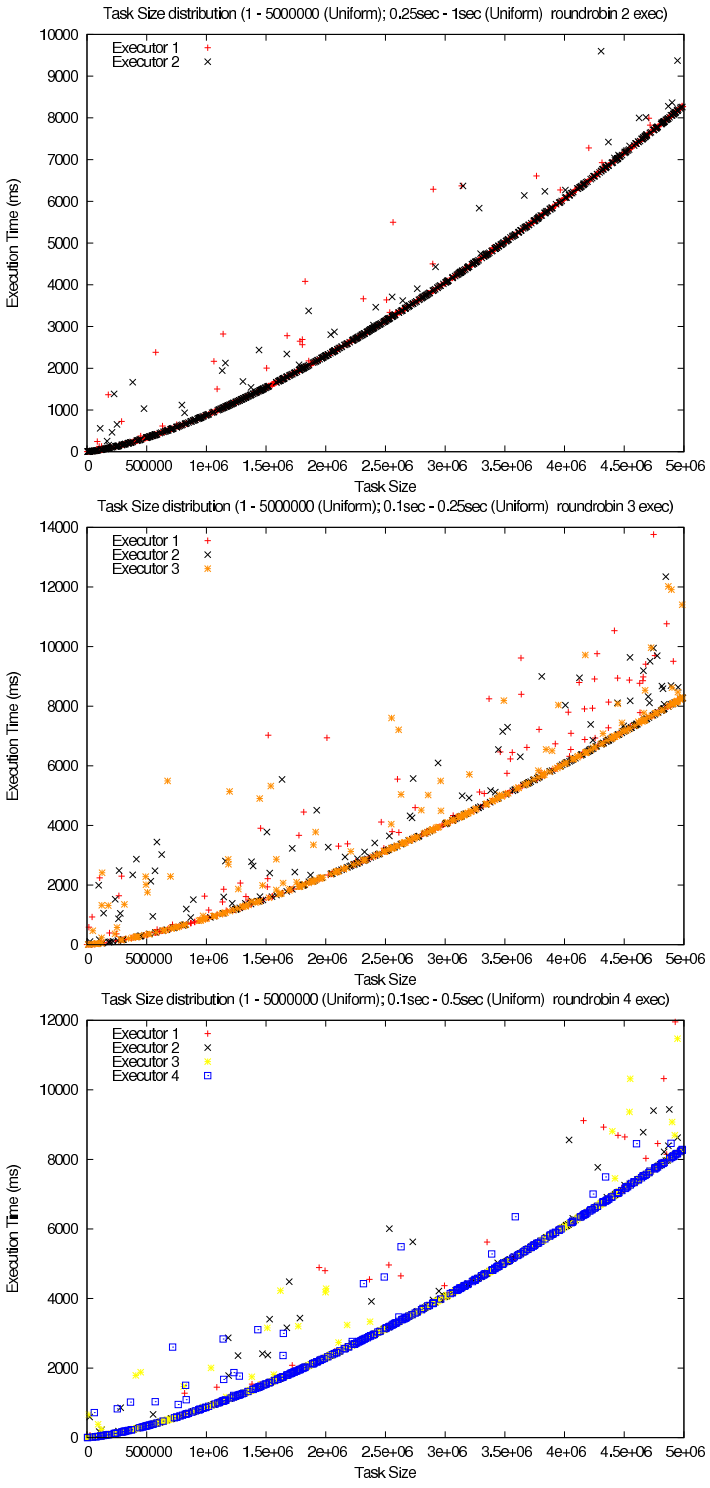

Figure 4.8: Task Size Distribution at Executors - RT-RoundRobin

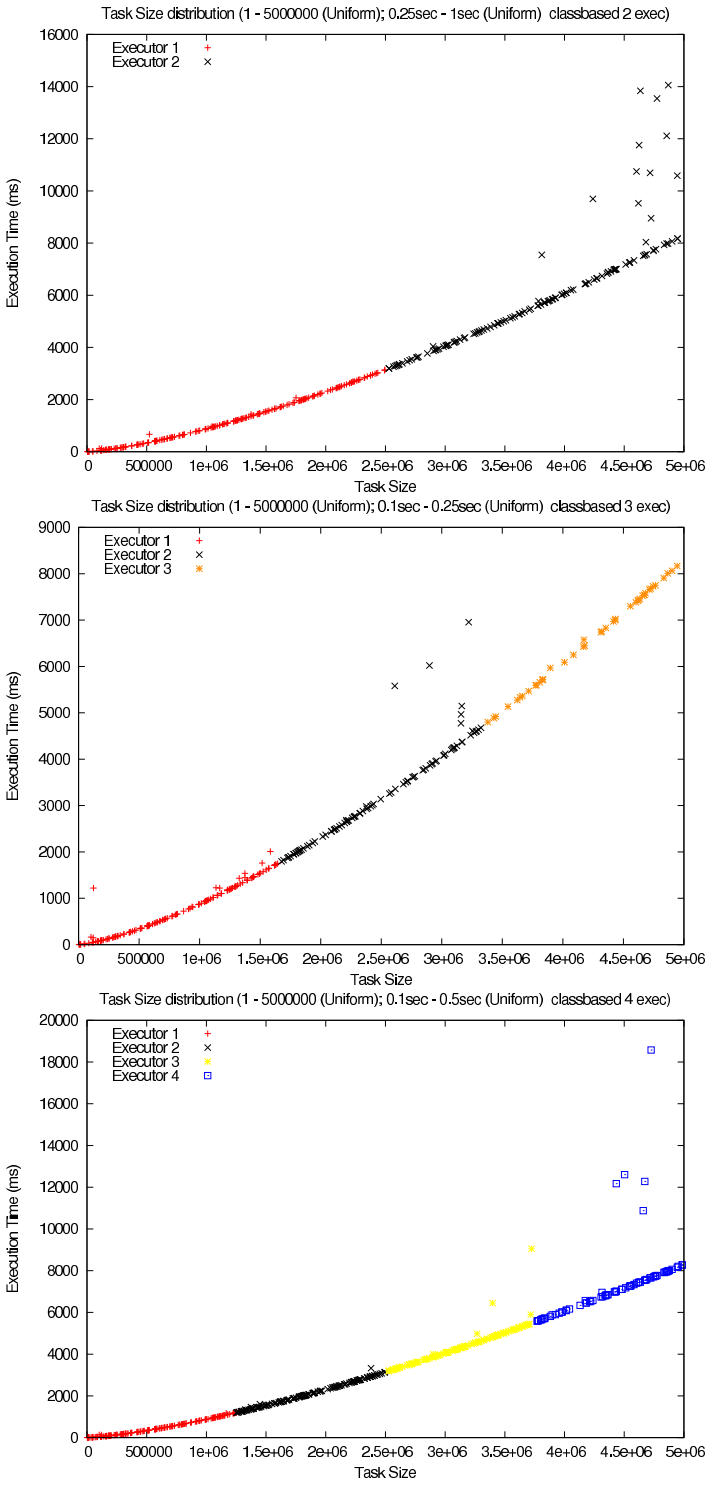

Figure 4.9: Task Size Distribution at Executors - RT-ClassBased

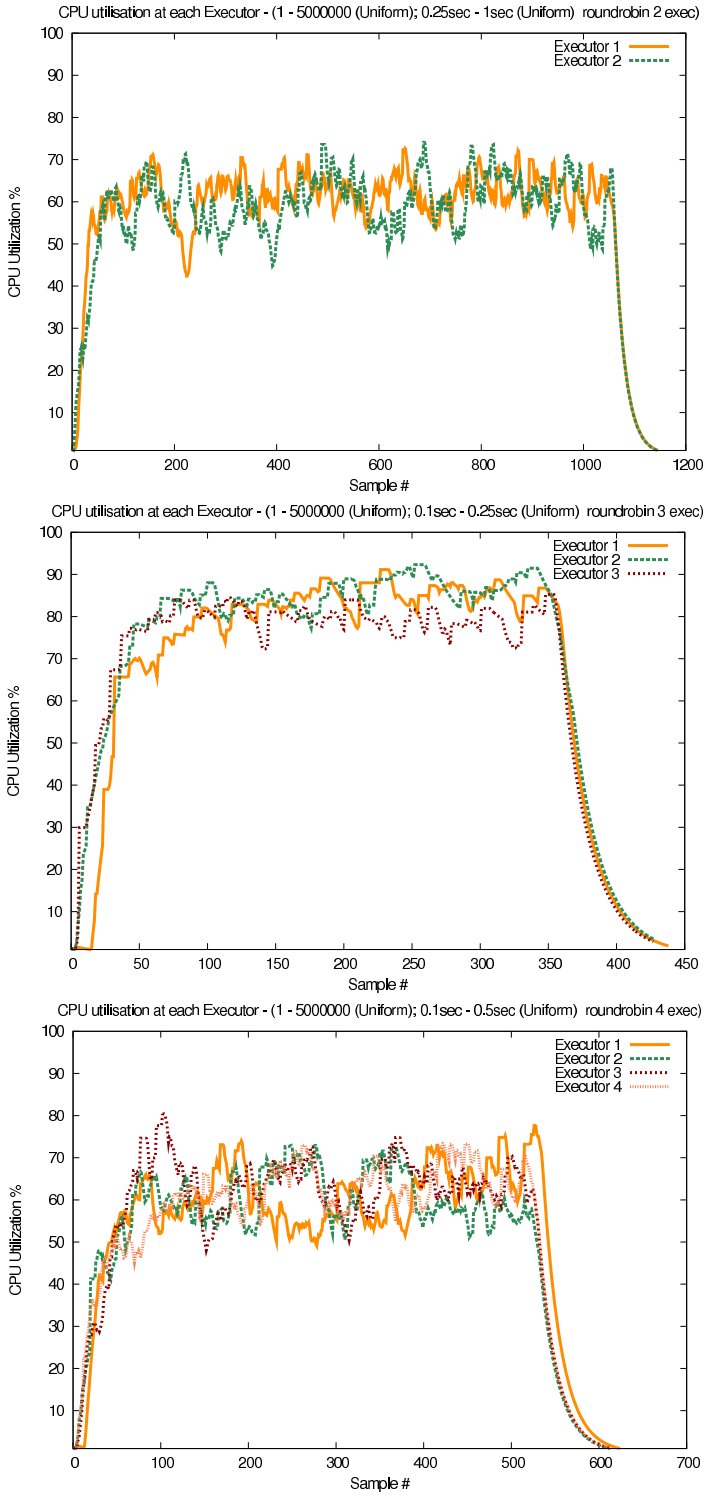

Figure 4.10: CPU Utilisation at Executors - RT-RoundRobin

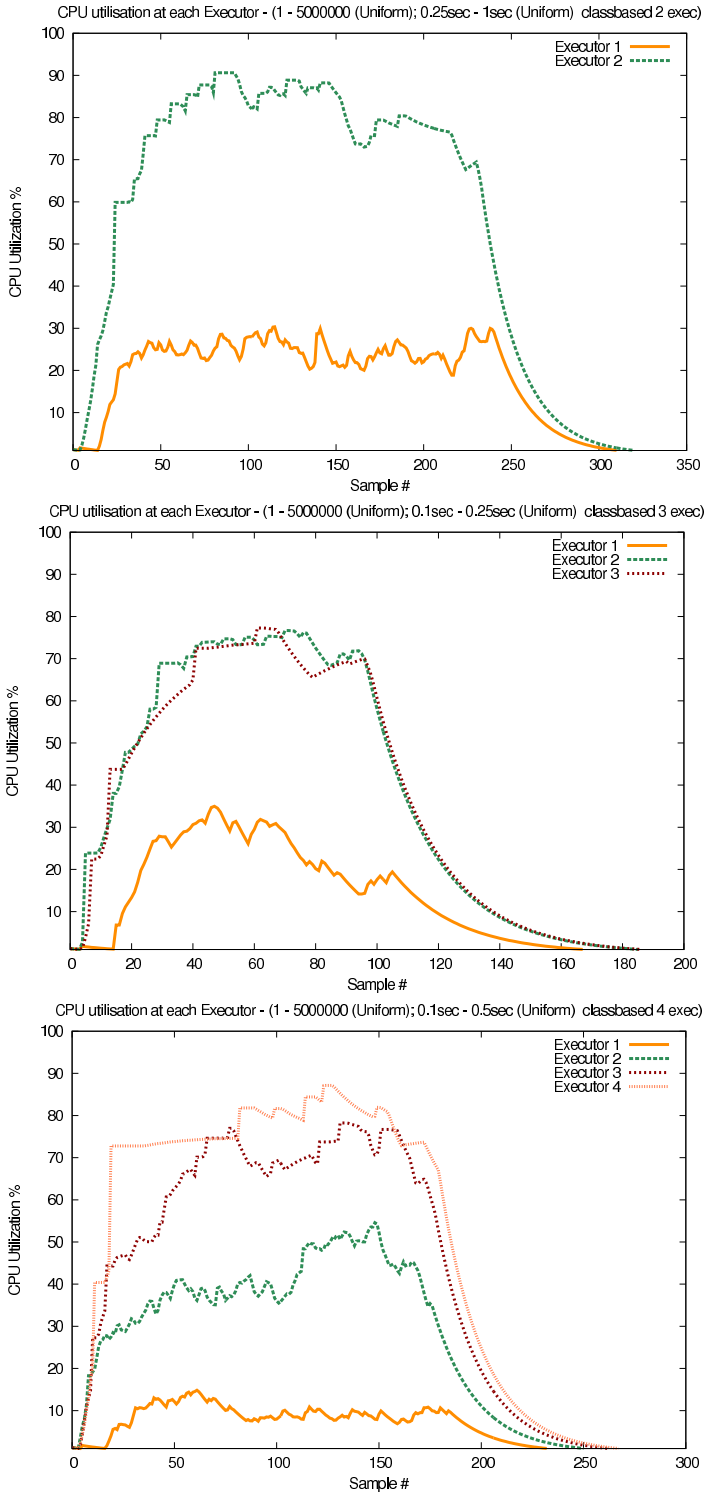

Figure 4.11: CPU Utilisation at Executors - RT-ClassBased

	Class Based (Non real-time)					
	2 Executors		3 Executors		4 Executors	
Inter arrival times (sec)	% Accept.	% Dead. Met	% Accept.	% Dead. Met	% Accept.	% Dead. Met
0.25 - 1	100	27.8	99.2	40.8	99.9	58.2
0.1 - 0.5	82.0	26.0	98.6	36.6	99.4	42.4
0.1 - 0.25	74.8	18.0	83.3	30.0	86.9	30.2
0.05 - 0.1	52.7	7.8	75.6	13.5	78.0	20.5

	RT-ClassBased					
	2 Executors		3 Executors		4 Executors	
Inter arrival times (sec)	% Accept.	% Dead. Met	% Accept.	% Dead. Met	% Accept.	% Dead. Met
0.25 - 1	99.2	99.0	100.0	99.0	100	100
0.1 - 0.5	62.2	95.4	76.7	94.8	90.9	100
0.1 - 0.25	45.4	94.6	66.0	99.0	74.4	97.7
0.05 - 0.1	28.6	98.9	44.7	91.4	55.1	99.0

Table 4.6: Performance Comparisons of Class based vs. RT-ClassBased

into classes based on such a condition. Such schemes try to achieve differentiated processing among these classes, where one might be favoured more than the others. As a result, many of such schemes may also result in unbalancing the load among cluster members. The feasibility of introducing the additional step of predictability based decision making into such schemes is investigated by RT-ClassBased algorithm.

In RT-ClassBased, we use request size to be the criteria for classification. Herein, segregation of requests based on size is a widely used technique that reduces the overall waiting time of the system. RT-ClassBased makes use of this feature whilst introducing the additional steps of selecting requests for execution based on their laxity and execution of requests based on their deadlines. In this evaluation, it is compared with a trivial class-based scheduling algorithm where each executor is assigned with a request size range. Table 4.6 contains the results while Figure 4.7 summarises them graphically (Note that the *x-axis* of the graph contains the mean value for the respective inter-arrival time period mentioned in Table 4.6).

The size based segregation of requests prevents scenarios where requests with a large size disparity compete for the same processing resource. Such scenarios would have requests with shorter execution times being queued behind requests with longer execution times. As observed from the results obtained, class-based scheduling perform better than round-robin scheduling due to this reason. Figure 4.8 contains plots for three different cluster configurations (2 to 4 executors) and arrival rates for RT-RoundRobin algorithm, illustrating distribution of task sizes among the executors. Figure 4.9 con-

	RT-LaxityBased					
	2 Executors		3 Executors		4 Executors	
Inter arrival times (sec)	% Accept.	% Dead. Met	% Accept.	% Dead. Met	% Accept.	% Dead. Met
0.25 - 1	99.2	99.9	100.0	99.9	100	100
0.1 - 0.5	89.0	99.8	80.5	99.8	99.8	100
0.1 - 0.25	47.4	99.2	66.0	99.6	75.2	100
0.05 - 0.1	38.5	99.0	50.7	99.2	54.3	100

Table 4.7: Performance of RT-LaxityBased

tains the plots for RT-ClassBased for the same configurations and task arrival rates. The difference in task size segregation among the cluster members by each scheduling technique is clearly visible. The dispersion of the execution times around each task size is less in class-based scheduling, as a result of the lower task size variance at each executor. Moreover, the dispersion of smaller sized requests are much lower in class-based scheduling (compared to round-robin) also due to the aforementioned reason.

Similarly, Figure 4.10 illustrates the resultant CPU utilisation levels for the same experimental runs when RT-RoundRobin is used. All three graphs for round-robin scheduling has all executors being utilised at similar levels. Figure 4.11 contains the CPU utilisation resulted by RT-ClassBased for the same experimental runs. In them, the different levels of utilisation at each executor due to the size based request segregation is clearly visible. Whilst simple class-based scheduling achieves better results than simple round-robin scheduling, RT-ClassBased performs even better when the percentage of deadlines met, are considered. Irrespective of scheduling decisions being made based on the size of requests, unconditional acceptance of requests and *best-effort* nature of execution may lead to overloaded conditions and requests being rejected. Moreover, the sharing of the CPU in *best-effort* processing prolongs the execution of all requests executing in parallel, thereby resulting in deadline misses. The schedulability check in RT-ClassBased coupled with deadline based scheduling, achieves more than 94% of the deadlines in any given scenario. Although acceptance rates are lower than simple class-based, the resultant deadline achievement rates clearly confirms RT-ClassBased outperforming its unmodified counterpart, in terms of predictability.

4.8.4 Laxity Based Dispatching

As discussed earlier, incorporating the schedulability check together with dispatching of requests, ensures requests are selected for execution based on their laxity property, after

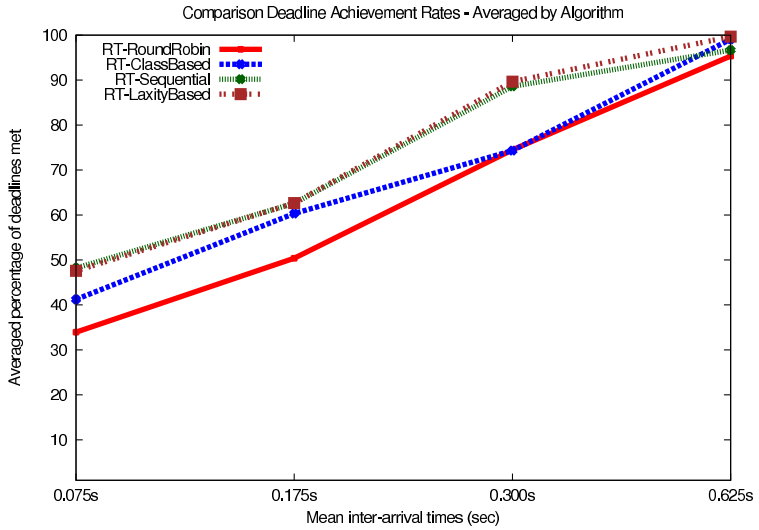

Figure 4.12: Deadline Achievement Rate Comparison of all Dispatching Algorithms

matching a request to an executor. The mapping of a request to an executor is decided by the dispatching algorithm used. In RT-RoundRobin the decision was to ensure equal distribution of requests among the cluster members and in RT-ClassBased it was to group similar sized requests together at each executor. RT-LaxityBased algorithm is a further step towards achieving better predictability by using the Laxity property even in the dispatching decision. Therefore, it demonstrates the predictability gain further achieved by using laxity based dispatching decisions.

RT-LaxityBased ensures the equal distribution of laxities among cluster members. This is an additional step to further ensure the larger range of laxities at each executor thereby enabling more requests to be scheduled together. Table 4.7 contains the results and Figure 4.12 compares the results with the other three dispatching techniques introduced. The selection of requests based on laxity and the additional step of distributing requests among executors based on laxity enables RT-LaxityBased to achieve better deadline achievement rates than the other three algorithms. Its features enables RT-Laxity to make use of the processing resources the best possible way. Compared to other policies such as RT-RoundRobin and RT-ClassBased, Figure 4.13 shows RT-LaxityBased resulting in equal utilisation levels for all executors in the cluster.

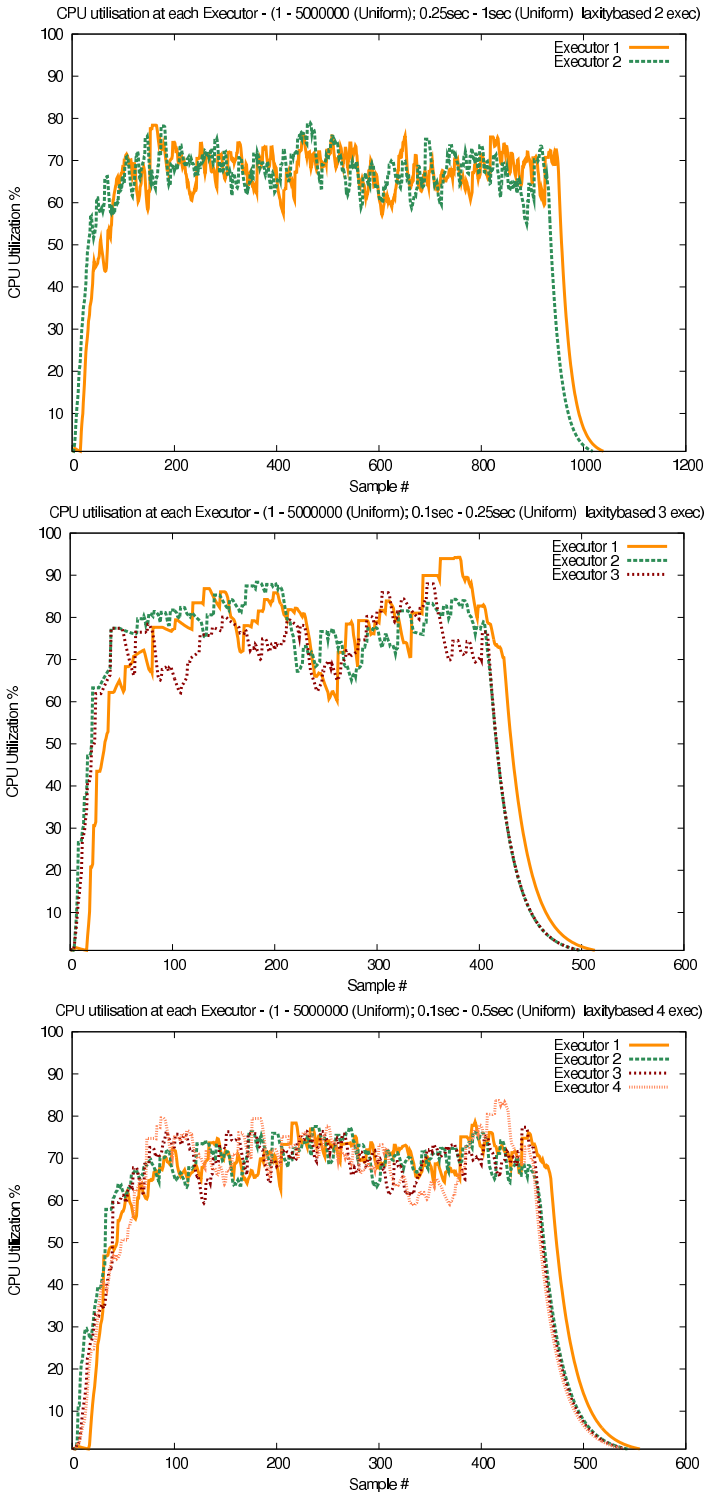

Figure 4.13: CPU Utilisation at Executors - RT-LaxityBased

	RT-Sequential					
	2 Executors		3 Executors		4 Executors	
Inter arrival times (sec)	% Accept.	% Dead. Met	% Accept.	% Dead. Met	% Accept.	% Dead. Met
0.25 - 1	99.0	96.8	100	97.0	100	97.2
0.1 - 0.5	86.0	91.0	96.1	96.3	100	95.0
0.1 - 0.25	38.6	87.4	76.5	95.0	84.6	96.2
0.05 - 0.1	29.1	90.0	57.2	95.3	66.7	95.8

Table 4.8: Performance of RT-Sequential

4.8.5 Exhaustive Dispatching

A common feature of the three dispatching algorithms presented earlier is the schedulability check with only the selected executor. While the single check was done to ensure the overhead by the schedulability check is kept to a minimum, checking schedulability of a request with more than one executor would increase the chances of the request being accepted. RT-Sequential algorithm is such an attempt to ensure a request is given the maximum chances of being scheduled on the cluster. Herein, a request is checked for schedulability with more than one executor until it is schedulable on one of them, or the list of executors are exhausted.

Table 4.8 contains the results when RT-Sequential was exposed to different request arrival rates at different in different cluster configurations. Figure 4.12 summarises these results and compares its performance with the other three algorithms presented. The persistent and exhaustive schedulability check in RT-Sequential results in highest acceptance rates of all algorithms. However, the total time taken for carrying out multiple checks can become significant for certain requests. Time accumulated by conducting multiple checks may lead to some requests missing their deadline. Therefore, RT-Sequential results in the lowest values for percentage of deadlines met out of all algorithms. However, given the better acceptance rates, the overall number of requests meeting their deadlines is only second to RT-Laxity Based. Nevertheless, the acceptance rates and deadline rates achieved were still better than the non-real-time algorithms we compared them with.

4.8.6 Laxity Based Request Selection

As previously discussed, laxity based request selection largely contributes to the success of achieving predictability through the algorithms outlined. Figure 4.14 depicts the resultant laxities by the admission control check when the cluster is running RT-

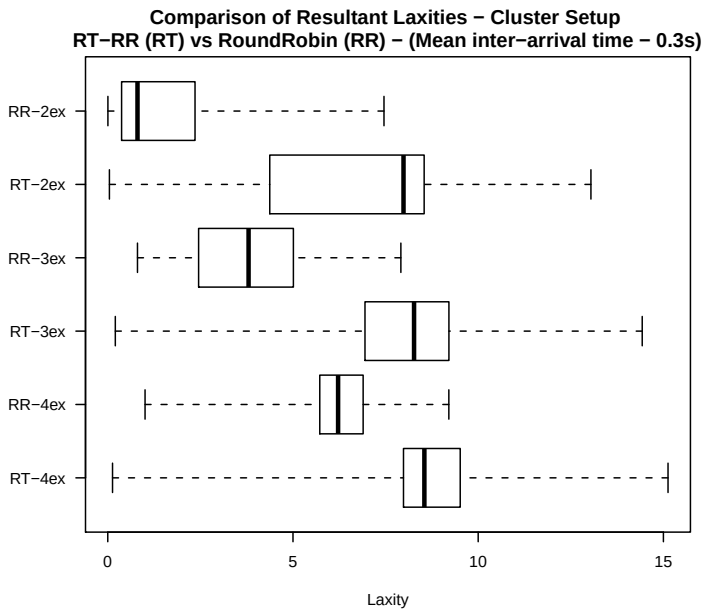

Figure 4.14: Comparison of Resultant Laxities from Admission Control

RoundRobin algorithm. Herein, resultant laxities for mean inter-arrival rate of 0.3 seconds when the requests are dispatched using RT-RoundRobin (RT) and Round-Robin (RR) are compared. Each plot corresponds to the average laxities achieved at the servers for each algorithm with 2, 3 and 4 executors respectively. Recall that request selection by the schedulability check results in a larger range of laxities at a server. Although similar patterns to the stand-alone results could be observed, the unmodified cluster running Synapse with Axis2 comparatively achieves a higher request acceptance and deadline rates due to the use of multiple executors, despite higher request arrival rates being used for the experiments. The two executor setup achieves the lowest median laxities due to the higher number of deadline misses of all runs. As the number of executors increase the miss rate decreases and the median laxity value increases. Moreover, the upper bounds achieved by the cluster setup is higher than the single server setup for obvious reasons.

The enhanced cluster setup with RT-Synapse and RT-Axis2 combination demonstrates a similar pattern in the resultant laxity values. Naturally, the selection process achieves a larger range of laxities as the median value decreases with high request arrivals. Although the increase of executors in the cluster does not result in a major change to the median laxity value, a shift in values from the lower quartile to the upper quartile is vis-

	Round Robin (Non real-time)		
	2 Executors	3 Executors.	4 Executors
Mean inter-arrival time	Reqs. sec^{-1}	Reqs. sec^{-1}	Reqs. sec^{-1}
0.625s (Low)	1.50	1.51	1.51
0.300s	0.81	2.91	2.92
0.175s	0.98	1.67	2.30
0.075s (High)	1.06	1.40	1.87

	RT-RoundRobin					
	2 Executors		3 Executors		4 Executors	
Mean inter-arrival time	Reqs. sec^{-1}	Reqs. sec^{-1} (excl. rejects.)	Reqs. sec^{-1}	Reqs. sec^{-1} (excl. rejects.)	Reqs. sec^{-1}	Reqs. sec^{-1} (excl. rejects.)
0.625s (Low)	1.62	1.42	1.62	1.61	1.62	1.62
0.300s	3.31	1.72	3.30	2.44	3.37	3.36
0.175s	5.54	1.55	5.46	2.56	5.43	4.24
0.075s (High)	10.3	2.11	10.8	4.05	11.1	5.13

Table 4.9: Throughput Comparison of Round Robin vs. RT-RoundRobin

ible. This is due to the constant rate of deadlines achieved by the setup ($\tilde{9}6\%$) and the increasing number of accepted requests being distributed to multiple executors in the cluster. From both configurations, it is clearly visible that such a purposeful selection of requests is a necessity for achieving predictability in execution.

4.8.7 Throughput Comparison

Next, we compare the unmodified and the enhanced versions of cluster configurations on their throughput. The unmodified cluster consists of Synapse and Axis2 where Synapse is configured to dispatch requests using simple round robin algorithm that it ships it by default. Note that both these products process requests in a *best-effort* manner and tries to execute all requests sent to it. The Axis2 executors are configured by default with 25 worker threads pre-created and to create up to 150 worker threads on demand. Similarly, Synapse is configured with 20 worker threads pre-created and can create up to a 100 as the queue fills up. Therefore, it is possible for the cluster to have 150 requests executing in parallel at each server. Similarly, dispatcher could process 100 requests in parallel.

Table 4.9 contains the results for the cluster running the two round-robin based scenarios for different executor configurations and the second graph of Figure 4.15 summarises them graphically. At the highest mean inter-arrival time (0.625s), both the unmodified and the enhanced cluster achieve similar throughput levels. As the mean inter-arrival

times decrease, requests arrive rapidly at the cluster. The unconditional acceptance of requests and *best-effort* processing results in the executors being overloaded. This makes them stall and be unresponsive to subsequent requests being directed at them. This makes requests time out and be rejected from processing. As the actual service execution takes place in the executors, they are the most affected by such conditions. As a request spends only a short time at the dispatcher, it is unaffected by such conditions and continues to dispatch requests to the overloaded executors. This unfavourable conditions in the unmodified cluster increases with the arrival rates of requests.

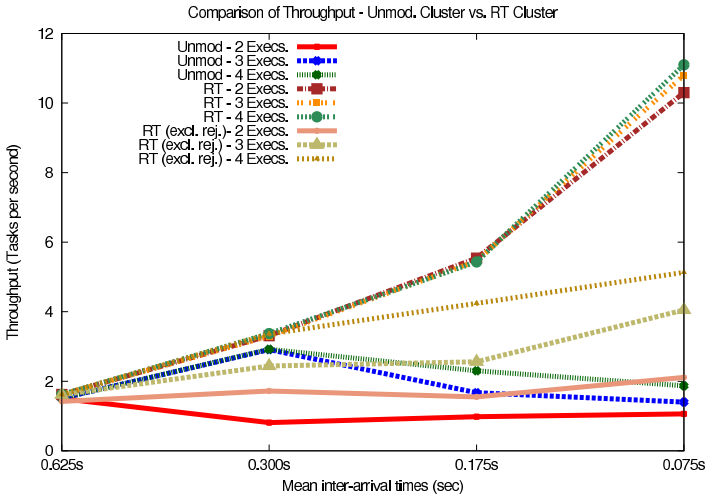

Figure 4.15: Comparison of Throughput Rates

The enhanced cluster setup with RT-Synapse and RT-Axis2 behaves differently to the unmodified cluster. The enhancements made to these products prevents the cluster from reaching overloaded conditions. As the mean inter-arrival times of requests decrease, the cluster shows increasing levels of throughput. However, the enhanced cluster completes processing of a request in two ways. Requests that are selected for execution, are completed at the executors and the result of the service invocation is channelled back to the client through the dispatcher. Moreover, a request that is rejected from the schedulability check is also considered as a completed request. These requests don't reach an executor as their processing completes at the dispatcher itself. Compared to the service invocation of a request, the time these requests spend at the dispatcher is very short. Therefore, the second column under each configuration for RT-RoundRobin contains a secondary throughput value calculated excluding the rejected requests. As expected,

the secondary throughput rate increases with the arrival rates. However, for the run with the highest arrivals, it can clearly be seen that nearly 50% of the requests processed are rejected requests. However, the system still achieves acceptable throughput rates while preventing it from reaching overloaded conditions. For a particular arrival-rate, observe that throughput does not change with the addition of executors into the cluster. While this adds more processing power to the back-end and leads to more deadlines being achieved, the throughput rate achieved by the dispatcher (be it Synapse or RT-Synapse) remains the same for the given mean inter-arrival time.

As RT-Synapse and RT-Axis2 is configured to have 3 lanes of execution, the real-time scheduler controls the number of threads in execution at a given time to just a single thread per lane at a given time. When processing workloads that are CPU bound using concurrency techniques, the optimum number of threads that will give the best performance would be equal to the number of cores available for processing [Subramaniam V., 2011]. Therefore, the enhanced cluster is configured for the processing resources at hand in an optimal manner. Despite having a lot more worker threads in the unmodified versions, it becomes inefficient due to the overloaded conditions it could lead to, when processing CPU intensive tasks.

4.8.8 Discussion

With the RT-RoundRobin evaluation, the benefits of the predictability enhancements in a cluster are directly visible as round-robin dispatching is a simple technique in its pure form. Moreover, it is also a perfect example of how a simple dispatching technique could be enhanced to achieve predictable execution. Similar to the stand-alone scenario, RT-RoundRobin clearly outperformed its unmodified counterpart. The effectiveness of round-robin scheduling is in the even distribution of requests it results in throughout the cluster. However, the same reason makes it unsuitable when predictability of execution has a higher importance. Even distribution of requests create a high variability of request sizes at each executor. Yet, this being a content blind dispatching technique, may result in higher loads and longer execution times due to the variability of request sizes. As the request execution happens in a *best-effort* manner with thread-pools executing as many requests as possible in parallel, all requests will complete with longer execution times albeit the throughput achieved (Figure 4.6). With RT-RoundRobin, the high variability of request sizes is circumvented due to the laxity based request selection in

the schedulability check. Irrespective of the size of the request, a request is selected for execution only if its laxity enables the deadline to be met while being scheduled with already accepted requests. Moreover, this also acts as an admission control mechanism that prevents server overloads. The high variability of laxities resulted by the schedulability check theoretically ensures deadlines of all requests accepted could be met, while deadline based scheduling ensures it practically.

RT-ClassBased algorithm serves as an example of how a request-aware scheduling policy could be enhanced to achieve predictable execution times. As with the other scenarios the RT-ClassBased algorithm outperformed the simple class-based version when deadline achievement rates are considered. As the size of a request was the criteria in matching a request to an executor, size based scheduling ensures that each executor is only faced with requests of similar sizes irrespective of the original task size distribution (Figure 4.9). This prevents chances of smaller sized tasks having longer waiting times due to the processing of a large sized request. Clearly, task-based scheduling in its pure form had better results compared to round-robin scheduling. RT-ClassBased combines this phenomenon with the additional guarantee of deadline requirement of requests being met. Yet again the difference is in the selection of requests based on laxity by the schedulability check and the purposeful scheduling of requests based on their deadlines. However, class-based scheduling in its pure form performs badly when percentage of deadlines met is considered. Similarly to earlier discussions, unconditional acceptance of requests and *best-effort* scheduling results in unpredictable execution times and deadlines being lost. RT-ClassBased is better suited for request streams with comparatively more smaller sized requests.

Experimental results confirm that RT-Sequential makes the best use of processing resources. It achieved the highest acceptance rates out of all algorithms. Trying to schedule a request repeatedly on different executors ensures that, a request will be scheduled on the cluster if required processing time is available on any one of the executors. This effectively fills the gaps on processor time lines making the maximum use of their processing resources. However, conducting multiple checks may incur a significant overhead depending on the size of the request. The life of a request, starts on its arrival at the cluster. Therefore, the time spent on being dispatched and being checked for schedulability, has to be subsumed within the execution time requirement of a request. For small sized requests, the overhead incurred by multiple checks may result in them missing their deadlines. As a result, RT-Sequential is not suitable for request streams

predominantly containing smaller sized requests.

The distribution of requests based on laxity, ensures that requests with large and small laxities are evenly distributed. If an executor gets too many requests with small laxities, eventually some of them will end up being rejected, as they compete for the same window of time. Requests with large laxities are able to shift or stagger their execution within a larger time window enabling more requests to be scheduled within their lifespan. This principle results in RT-LaxityBased meeting the highest number of deadlines, with more than 50% acceptance rate in most cases (second row of Figure 4.12). All algorithms show that they could achieve higher performance with the cluster scaling up with more executors. With cost of hardware becoming cheaper by the day, the acceptance rates could be increased with more executors being added to the cluster. In such a setup RT-LaxityBased will be the best algorithm to use with a good mixture of task sizes and laxities in the request stream.

From the results presented, it can be observed that the inter-arrival times of tasks affect the request acceptance rates and deadlines met. When arrival rates are increased, requests arrive far more rapidly at the cluster. Due to more requests competing for the same window of time, the schedulability check results in a higher rejection rate. Similarly, the number of executors in the cluster affects the request acceptance and deadline achievement rate. More executors in a cluster would mean having more processing resources for request execution. The distribution of requests among a larger number of executors would create reduced arrival rates at each executor. This allows more requests to be scheduled within the cluster, resulting in higher acceptance rates. Though having additional executors would seem to be more processing for the dispatcher, the impact is not significant when the worst case time complexities of the scheduling algorithms are considered. Moreover, this enables the cluster to scale without a cost on the processing at the dispatcher.

The role of laxity in achieving predictability and its importance can be observed in the results discussed in the laxity comparison. While *best-effort* processor sharing execution is ideal for common processing tasks, ensuring predictability mandates a suitable method of admission control that contributes towards the goal. Request selection based on laxity gives an assurance of meeting a request deadline even prior to its acceptance for execution. The wide range of laxities achieved by the selection process ensures that requests with complementing laxities execute successfully within a given window of

time.

The throughput achieved by the enhancements indicates that its performance is comparable with the unmodified version, in low traffic conditions. Although the enhanced version outperforms the unmodified versions in high traffic conditions the higher throughput values are largely contributed more by the request rejections. However, when throughput is calculated excluding the rejections both configurations still achieve acceptable throughput rates with resilience to high traffic conditions, contributed by the admission control mechanism. While the unmodified versions succumb to system overloads, they are bound to perform better than the enhanced versions in favourable conditions. Another aspect considered is the nature of tasks handled by the two system. In order to have control over the task sizes of requests the web services used for testing created CPU bound work for the servers. However, for services that is more I/O bound, the configuration of the unmodified middleware maybe more suitable. Therefore, the enhanced versions can only be considered resilient to high traffic conditions. Considering them to have better throughput values under normal conditions is unfair on the unmodified versions of the products.

4.9 Summary

In this chapter we presented four request dispatching algorithms intended to be used in clusters hosting web services, to achieve predictability of service execution. The mathematical model and the schedulability check presented in chapter 3 is incorporated into these scheduling algorithms to choose requests for execution based on their laxity property. The algorithms match a request to a dispatcher in different ways and would check for schedulability with one or more executors. This laxity based selection process ensures that selected requests have complementing laxities to that of already accepted. Through this process the dispatcher ensures the required deadline of the request can be met while ensuring deadlines of already accepted requests are not compromised. The selected requests are executed by the servers using EDF scheduling principle.

With RT-RoundRobin we presented how a simple request agnostic algorithm could easily be enhanced to support predictability. Similarly, RT-ClassBased was presented as an example of an enhanced request-aware dispatching algorithm. Moreover, RT-Sequential algorithm was designed to make best use of cluster resources where it gives the best guarantee of scheduling a request on the cluster. Finally, RT-LaxityBased algorithm

was presented as a method of extending the laxity based request selection into laxity based dispatching of requests, by which it tries to maximise the range of laxities at a server. These algorithms were implemented in a real-life cluster using Apache Synapse and Axis2. These middleware products were enhanced to include the dispatching algorithms to support request deadlines and real-time scheduling. The predictability gain by the enhancements were measured by comparing the enhanced cluster to a cluster using unmodified versions of Synapse and Axis2 in their default configurations.

The empirical results indicated that the cluster is able to achieve acceptable levels of predictability of service execution while maintaining satisfactory throughput rates. Moreover, the enhancements make it resilient to high traffic conditions and prevents the system from reaching overloaded conditions. Apart from the algorithms introduced, the implementation level enhancements made to RT-Synapse and RT-Axis2 played a major part in the predictability achieved by the cluster. A detailed discussion on the software engineering techniques used in the implementation is presented in chapter 5.

While the predictability gain was measured in terms of the number of deadlines achieved, an important aspect introduced to the middleware as part of the enhancements were the differentiated service processing. This was primarily achieved through the deadline based scheduling done by the real-time scheduler. In RT-ClassBased we saw a typical class based request classification being followed and supported by deadline based scheduling. In the next chapter we investigate deadline based scheduling further and try to obtain an advanced model using queuing theory that allows us to analyse and predict the behaviour of a system using EDF scheduling policy.

Chapter 5

Building Web Services Middleware with Predictable Execution[§]

This chapter presents software engineering techniques, algorithms, designs and tools that are geared towards achieving predictability of execution in web services middleware. Conventional designs and development techniques can have a negative impact on predictability of execution. For instance, throughput is considered as a major design goal in engineering web services middleware. Techniques employed to achieve throughput such as unconditional acceptance and *best-effort* processor sharing execution of requests result in unpredictable execution times. Moreover, conventional debugging techniques can lead to priority inversion scenarios. Engineering systems for predictability requires a different way of thinking in building systems. Completion of requests within a perceived deadline require them to be explicitly scheduled, and the middleware must have better control over their execution. As such, the proposed designs change the execution within web services middleware to be more serialised, while achieving controlled level of throughput by limiting the number of concurrent executions, based on the number of processor cores available. This is made possible by a few techniques introduced. Firstly, the conventional thread-pools in web services middleware are replaced with custom designed ones using real-time threads. The best-effort processing of the middleware is replaced by a priority model that gives more control over the execution and suspension of worker threads. A newly introduced real-time scheduler

[§] Preliminary versions of the work presented in this chapter have been previously published in [Gamini Abhaya et al., 2010b, 2012].

component uses the priority model to control the execution of worker threads. The use of development libraries and operating systems with predictability features, empower them with increased control over execution. Specialised debugging techniques such as in-memory logging, delayed writes and the use of specialised tools such as the Oracle Thread Scheduling Visualiser used for offline debugging, are introduced to prevent unnecessary priority inversions by debugging the system. The successful application of these techniques and tools, are presented using two case studies, in which the functionality of Apache Axis2 and Apache Synapse are enhanced to represent stand-alone and cluster-based web service deployments. Moreover, the four dispatching algorithms presented in the previous chapter are implemented using the sequence and endpoint framework in Apache Synapse. A detailed discussion on the designs, techniques and tools used is provided as part of this chapter. Furthermore, a generic set of guidelines that summarise the important milestones in achieving predictability of execution is also presented, to be used in identifying predictability features in existing middleware and to decide on the enhancements necessary.

5.1 Motivation

The design and architecture of web services middleware plays an important part in the performance of web services it hosts. In service management and invocation, the middleware carries out the house-keeping tasks required, such as request processing, request execution, metadata management and error handling. Such middleware are designed with the intention of achieving high levels of throughput to maximise the processing of requests. The level of throughput achieved by these middleware comes at the cost of unpredictable execution times for service invocations. Typically, service received by the tasks are inversely proportional to the number of jobs present in the middleware [Coffman Jr et al., 1970; Stantchev, 2009; Subramaniam V., 2011]. While non-critical operations such as a WSDL request may not be affected by this phenomenon, it is of concern for achieving perceived levels of execution time predictability in web services middleware.

The concepts, algorithms and scheduling techniques discussed in the previous two chapters can only be put to use by implementing them in web services middleware. In doing so, they needs to be supported by many implementation decisions. For instance, the execution deadline needs to be conveyed to the middleware in some manner and be

available for the admission control check. Similarly, the deadline and laxity information need to be made available to the dispatching algorithms. The implementation of EDF scheduling would require the ability to control the execution of worker threads at runtime. As such, the decisions made in implementation is equally important as the techniques and algorithms presented in the earlier chapters.

Previous successful attempts at achieving predictability of execution in other distributed communications technologies can be seen in the work of Schmidt et al. in their work on Real-time CORBA [Pyarali et al., 2003; Schmidt and Kuhns, 2000; Schmidt et al., 1997; Wang et al., 2000]. They used real-time scheduling algorithms, custom made networking protocol stacks, customised request executors, an end-to-end priority model that is used by the executors, that enable more control over executions that happen through the CORBA middleware. Compared to web services, CORBA is used within a comparatively static environment (typically on a Local Area Network) where request properties and communication flows are known *apriori*. While the work on CORBA signifies achieving predictability is possible even in web services, the environment they operate in is far more challenging.

While the best possible solution would be to build a middleware from ground-up optimised for predictability, it was not the most suitable given the time constraints, this research had to be conducted within. As developing a complete middleware ensuring predictability requires both time and effort, it was deemed too much for a single researcher to complete within four years allowed for the degree. Therefore, an alternate solution (despite being sub-optimal) was sought for, by enhancing existing web services middleware to have predictability features. Such an attempt not only makes it feasible to achieve a working solution within the time frame, but ensures existing web service deployments can benefit from it.

5.2 Problem Statement

Web services middleware contain many techniques to maximise service invocations. To achieve better rates of throughput they are designed to accept all incoming requests and execute as many requests as possible using *best-effort* processing. Concurrent execution of requests is facilitated by employing thread pools [Graham et al., 2004] using processor-sharing execution. From a predictability standpoint this method does not scale-up well as the increasing number of requests handled by the middleware

results in inversely proportional execution times [Coffman Jr et al., 1970; Stantchev, 2009; Subramaniam V., 2011]. Moreover, the mean execution time of a task varies with the number of requests handled by the middleware. As a result, the execution of two service invocations, despite using the same input parameters may result in vastly different execution times.

Unconditional acceptance of requests ensures the middleware handles as many requests as possible. Concurrent execution of requests are only limited by the number of worker threads active in the thread pool and processor sharing execution ensures that every request is attended to as soon as they arrive at the system. Similarly, best-effort processing means that there is no differentiation between the requests being executed. While employing all these techniques maximises the throughput rate the middleware achieves, it leaves the middleware with no control over the execution of requests which results in unpredictable execution times.

Predictability of execution cannot be guaranteed by the middleware merely from its functionality. It needs adequate support from the development platform, libraries used and the operating system. For instance, standard Java has 10 priority levels and they are not strictly enforced as they do not map directly onto operating system level priorities [Bruno and Bollella, 2009; Dibble, 2002; Oracle Corporation, 2009a]. Therefore, the execution of a standard Java thread assigned with the highest priority available, can be interrupted by a thread or process outside the Java virtual machine running at a higher operating system level priority. Moreover, languages such as Java have specialised processes (called garbage collectors) to reclaim memory from expired objects. They operate on special priorities that could interrupt the execution of any priority level available in standard Java [Bruno and Bollella, 2009]. Such instances would add unwarranted delays to the execution of tasks in web services middleware.

Based on the above, the research question addressed in this chapter is *"How to build web services middleware with predictable service execution?"* In finding solutions to this question, we identify three main problem areas in existing web services middleware that contribute towards unpredictable execution times, we address in this chapter.

1. Design strategies used for achieving throughput having a negative impact on predictability of execution.

2. Unsuitability of trivial development and debugging techniques.

3. Lack of development platform and system level support for predictability.

5.3 Outline of the Solution

Engineering web services middleware for predictability requires a different approach to engineering for throughput. The overall aim is for the middleware to have more control over the execution of requests. To enable this, the execution of requests is serialised by scheduling requests for execution one after the other in the order of their increasing deadlines. However, a controlled level of throughput is still achieved by implementing several lanes of execution where multiple worker threads are be active at a given time, depending on the number of processor cores available for execution. The conventional thread-pools in the middleware are replaced by custom designed thread pools that employ real-time threads. More control over their execution is further facilitated by introducing a multiple priority model, containing priority levels that cannot be interrupted by housekeeping activities and other processes. Such priorities ensure that the execution of a thread happens uninterrupted. The priority model is used by a real-time scheduler component to execute and suspend worker threads at will.

Moreover, EDF scheduling is implemented within the real-time scheduler in the form of a thread scheduling algorithm. Such fine grain control over thread execution requires the introduction of proper concurrency control using critical sections and semaphores, over the implementation of the admission control check, EDF scheduling and request dispatching in the clusters. The execution deadlines are conveyed to the server using SOAP message headers, and they are conveyed to the admission control check through an internal data structure. These implementation decisions are supported by the use of development platforms, libraries and operating systems that contain real-time features.

The proposed solution involved implementing these techniques in two web services middleware products, namely Apache Axis2 and Apache Synapse. Axis2 is a stand-alone middleware and Synapse is an ESB product that can be used to implement dispatcher functionality in a cluster. Two case studies based on the enhancements carried out on these products are presented in this chapter as practical examples for achieving predictability of execution. One case study represents a stand-alone web service middleware configuration using Apache Axis2 and the other discusses a web services cluster deployment using Apache Synapse as the dispatcher and Axis2 instances as executors. Three possible cluster configurations are presented and their pros and cons are

discussed. The four dispatching algorithms are implemented in Apache Synapse using its sequence and endpoint extension framework.

The contribution through this chapter are the specialised software engineering techniques, algorithms, designs and tools that can be used for achieving predictable execution times. The two case studies are examples on how they could be used in enhancing existing web services middleware. Although specific products were chosen for these case studies the techniques, algorithms and designs are generic enough to be applied for any other web services middleware.

The rest of the chapter is organised as follows. First, we discuss some of the related work found in this area. Next, we present the set of guidelines to follow to achieve predictability of service execution, when enhancing existing middleware products or new ones being built. Thereafter, in Section 5.6 we give an overview of the implementation and discuss items common to both implementations. Following that in Section (5.7), we present the first case study of enhancing Apache Axis2. It is followed by the case study of the cluster-based implementation and then we conclude in Section 6.9.

5.4 Related Work

There has been only a few attempts at introducing predictability into service execution or making it a feature in the middleware. As mentioned earlier, many of the existing middleware products optimise for throughput rather than predictability and thereby introduce features that makes it impossible to achieve predictability of execution. A few of the existing work we found that were purposefully built to differentiate request processing or to have real-time features, are discussed here.

wsBus [Erradi and Maheshwari, 2005] is a custom built QoS-aware middleware based on a bus architecture. It has many components that facilitate the use of different transports, request dispatching, service selection and QoS monitoring and has the design of a customised ESB product. It supports the use of priorities for differentiating requests and contains an admission control mechanism that controls requests accepted. However, the criteria of admission nor the prioritisation has been explicitly mentioned. Similarly, the way differentiation happens and any evaluation of its performance has not been presented.

The attempt by [Helander and Sigurdsson, 2005] to use SOAP based web services in an

embedded real-time environment is the first attempt we found in literature of such middleware. Web Services are used for communication between different components in the embedded environment. They achieve this by defining behavioural patterns among the components that represent interactions between components and tasks that need execution as a result. The timing properties, worst case resource requirements of these patterns are figured out at design time of the system. At runtime, these patterns are used to predict and reserve resources for the incumbent tasks. The worst case resource requirements planned at design time ensures that variations in execution and jitter are catered for, by over reserving resources at run-time. However, neither an architecture, implementation nor an evaluation has been presented in the paper.

[Mathes et al., 2009a] presents SOAP4IPC: a real-time SOAP engine designed for industrial automation. It contains general components as found in a typical SOAP engine that takes care of processing and execution of web services and also components that are designed to represent tasks typically found in real-time systems (as presented in chapter 2). The execution engine honours a deadline and caters for both periodic and aperiodic tasks. Given the support for periodic tasks and the not having an admission control check means the middleware is intended to be used with tasks, that properties are known at design time. An approach suggested by the authors is to use a profiling approach of measuring the worst-case execution times of each service. However, there is no mention of the actual scheduling algorithm used or a comprehensive evaluation with realistic services and traffic types. The SOAP4IPC engine is a part of a broader framework named TiCS [Mathes et al., 2009c] which stands for Time-Constrained Services framework that is presented as a complete manufacturing execution system that uses web services for industrial automation. The SOAP4IPC engine is at its core handling the execution of services and another layer which is detailed in [Mathes et al., 2009b] works as a service façade for programmable logic controllers that make up the manufacturing layer or the overall manufacturing execution system. The TiCS framework is described in detail in terms of its components and their intended functionality, however there is no mention of the actual scheduling algorithm is used or how the deadlines are ensured by the system.

While [Helander and Sigurdsson, 2005; Mathes et al., 2009a] and [Mathes et al., 2009c] are mentioned as working solutions in real-time environments they lack important details such as how the requests are scheduled and a comprehensive evaluation of their performance compared against other products and approaches, to make a proper deci-

sion on their effectiveness. Moreover, both these solutions are intended to be used in closed environments where there is a good understanding of task properties and their resource requirements. Without having proper admission control and precise scheduling, it is difficult to use their techniques to introduce predictability into web service execution in open systems. The challenge of selecting tasks based on resource availability and making that decision at runtime is an important aspect that has to be met with such open systems.

5.5 Guidelines for Achieving Predictability of Execution

Complex software follow modularised designs aimed at maintainability and reuse through shared libraries. Similarly, web services middleware consist of a collection of software components that make use of functionality provided by various development platform libraries and OS level services.

As illustrated in Figure 5.1, web services deployed within a server are exposed through the middleware to the outside world. The middleware handles all requests (SOAP and REST) with the aid of many development platform libraries that handle message processing and network level communication. The functionality provided by the development libraries are facilitated by the underlying OS. The OS handles the execution of threads, processes and manages system level resources such as CPU time, memory, sockets for network communication, access to Input/Output devices and other peripherals. For managing resource allocations, execution of processes and threads, the OS uses system level priorities for differentiation. These priorities, can be requested by the development platform or defaulted to OS preferences. The OS decides on the precedence of execution and resource allocation based on such priorities. Therefore, any form of predictability at the upper layers of a software stack, is only achieved with the support of all underlying layers.

From our research into the design of real-time systems, we present the following guidelines that would enable web services middleware to achieve predictability of execution. At a high level, these cover functional as well as aspects of software engineering aspects that could be used to enhance existing web services middleware, or when they are newly developed.

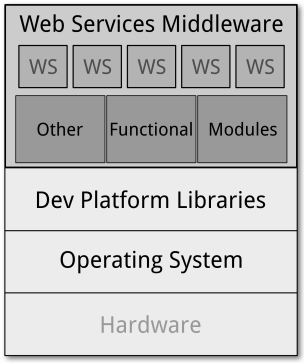

Figure 5.1: Default Software Stack

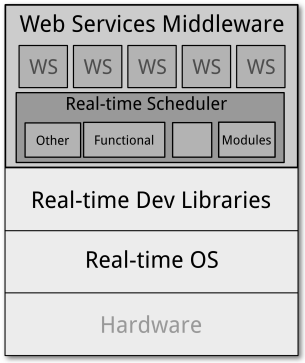

Figure 5.2: Required Software Stack

G1. **Use of an operating system, development platform and libraries with predictability features.**

Predictability of execution in a server can only be achieved if such features are provided by the lower layers of software being used. Most widely used development platforms and operating systems are intended for general use, thus have no support for predictable execution. For instance, thread priority levels used in the standard and enterprise versions of the Java development platform do not directly map to the range of priorities available at the OS level [Oracle Corporation, 2009a]. As a result, the execution of a Java thread running at the highest priority available in Java, can be interrupted by other processes running with higher OS level priorities. Similarly, it could also be interrupted within the platform itself, by housekeeping activities such as garbage collection [Arnold et al., 2006]. The use of specialised real-time development platforms and OSs ensure predictability by having features such as high precision clocks, fast context switches with minimum overhead, guaranteed priority levels, fast memory based I/O, faster responses to interrupts and priority inheritance mechanisms [Stankovic and Rajkumar, 2004]. Figure 5.2 depicts such a setup with the software required in all levels.

G2. **Support deadlines for service execution and decisively schedule requests to meet them.**

The invocation of a web service typically happens semantically equivalent to a method call of an object, where input parameters are specified and a result is returned. For the invocation to always complete within a target, web services middleware must be specified with a time limit. Therefore, middleware supporting predictability of execution must introduce means of specifying a user perceived deadline. Subsequently, the middleware must explicitly schedule the service invocation to meet the specified deadline. However, in the event of multiple invocations having overlapping executions, the middleware must ensure that scheduling of a request based on its deadline does not compromise the others with overlapping lifetimes.

Figure 5.3 depicts a schedule of tasks based on their deadlines. Herein, tasks are executed in the increasing order of their deadlines and completes execution in the order of T3,T4,T5,T2 and T1. On its arrival, each task has an overlapping lifespan with one or more tasks already in execution. However, tasks with earlier deadlines have been able to finish their execution within the required time limit, as a result of being explicitly scheduled on this basis. Although having arrived at the system later, tasks T3, T4 and T5 have been explicitly scheduled to complete prior to T1 and T2 by preempting them from execution. Subsequently, T1 and T2 also achieve their deadlines despite being executed in a staggered manner due to their large deadlines.

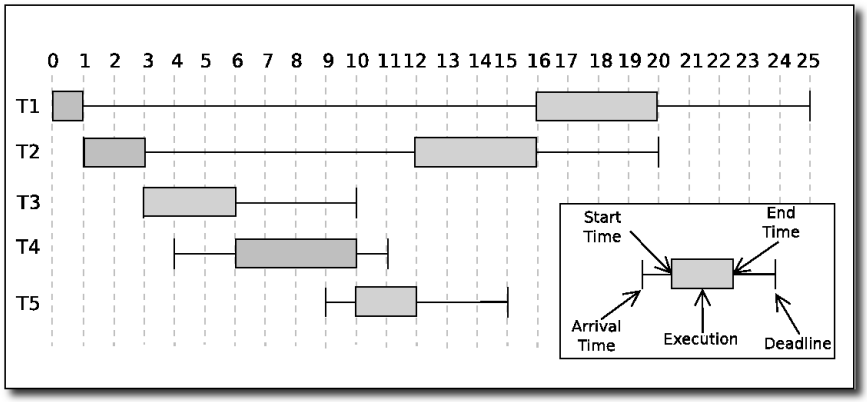

Figure 5.3: Deadline based task schedule

G3. **Conditionally accept requests for execution based on their laxity property.**
The lifetime of a task arriving at a system is determined by its arrival time (assuming it is ready for execution) and the perceived deadline, before which it must complete execution. Depending on arrival rates of requests, it is quite common for them to have overlapping lifespans with each other. While scheduling these tasks with a deadline guarantee, maybe possible by delaying the execution of unfinished tasks with longer deadlines, there will be instances where staggered execution of tasks would not be possible without a deadline miss. However, the deadline of an already accepted task should not be compromised for a new task even though it maybe having an earlier deadline. Therefore, it is imperative that requests must be accepted for execution conditionally, ensuring deadlines of other requests are not compromised.

The laxity property of a request is a measurement on the possible delay of execution while meeting the deadline requirement. A larger laxity enables the execution of a request to be delayed safely, thereby allowing more requests to be scheduled together. Scheduling a given set of requests ensuring their deadlines, is only possible with a greater range of laxities within them. For instance, the deadlines of tasks T3, T4 and T5 in Figure 5.3, have been achieved due to the larger laxities that resulted in the delayed and phased out execution of tasks T1 and T2. Similarly, the shorter laxities of T3, T4 and T5 enabled them to achieve their deadlines within the lifespan of T1 and T2. Conversely, T5 may not have been able to achieve its deadline executing together with T3 and T5, if it had a smaller laxity. Therefore, requests must be consciously selected for execution, resulting in a large range of laxities at the server.

G4. **Achieve differentiated request processing at system level.**
The invocation of a web service has many steps to be fulfilled by different components inside the web services middleware. Common to any such middleware, the execution of a request is typically handled by one or more worker threads (the smallest unit of execution) throughout its entire lifetime within the middleware. While the execution times at each component may vary depending on the nature of processing, the individual times are subsumed within the overall execution time of a request. Widely used web services middleware treats all threads equal and makes no differentiation in their processing. This results in the middleware

having no control over the completion time of a request.

However, achieving predictability in execution is only possible, if some differentiation in request processing is achieved within the middleware. For instance, when a new task with an earlier deadline arrives at the system in Figure 5.3, the execution of the current task has to be suspended and resumed at a later point of time. It must be possible for the server to suspend the execution of one task (e.g. T2 or T1) and let another start execution (e.g. T3). Therefore, at any given time the middleware must be able to control which thread is in execution and which is suspended. This fine-grain control will allow the middleware to decide on how the processing resources are consumed by the smallest units of execution. Such control will enable the middleware to avoid deadlocks and unnecessary delays on execution due to resource unavailability. A properly managed set of priorities makes it possible to achieve such fine grain control over the execution of threads.

G5. **Reduce instances of possible priority inversions.**

Contention for system resources is often encountered in task execution. Another form of delay that maybe added to the execution of a request is the possibility of a priority inversion. This refers to the scenario where a resource required by a higher priority process or a thread is held by a lower priority process or thread [Stankovic et al., 1998]. As depicted in Figure 5.4, this could take place when the lower priority thread in execution that was consuming resource X is preempted by a higher priority thread. The higher priority thread also wishes to consume resource X to complete its execution. However, this becomes impossible as the resource is currently held by the lower priority thread which has been preempted from execution; and a deadlock arises. The hold on the resource by the lower priority thread may finally be released by a time-out, if such a mechanism is used by the OS to free unreleased resources. In which case, the high priority thread maybe able to resume execution, albeit the delay incurred by the wait.

Real-time OS design often solve such problems using priority inheritance algorithms [Stankovic et al., 1998]. Consumption of resource X is a necessity for the higher priority thread to complete its execution. Since its execution cannot be resumed till resource X is released, the OS makes the lower priority thread ‘inherit’ the higher priority temporarily, to resume its execution and release the resource. Once the resource is released, the priorities are inverted back to their original

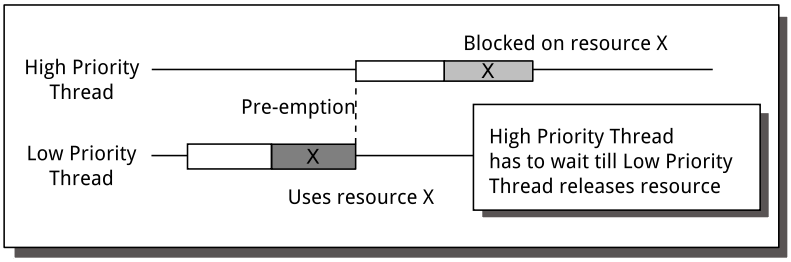

Figure 5.4: Priority Inversion

state and the execution is resumed. Although this mechanism solves the problem, it adds an unwarranted delay (equal to the execution of the priority inheritance algorithm and the controlled execution of the lower priority thread) to the overall completion time of the high priority thread. Such priority inversions may happen with simple I/O operations such as writing to a file or displaying a message to the console. Moreover, they would not only create delays in the actual execution, but be responsible for unexpected results and behaviour in activities such as debugging such applications. For instance, common debugging practices such as the use of log files and trace messages can result in unexpected priority inversions. Prevention of such phenomenon is only possible by avoiding such trivial techniques and using specialised ones instead.

Adhering to these guidelines specified above, will enable web services middleware to function with predictable execution and be successfully built accordingly. While these are valid for both SOAP and REST based web services, the way they are implemented in various middleware, may differ from each other. In the case studies presented in the next section, enhancements made to two widely used web services middleware are presented. Although these were used as examples, the enhancements are generic enough to be applied for any other middleware product available.

5.6 Implementation Overview

This section presents the preliminaries for the enhancements made to the stand-alone and cluster-based web services middleware. Enhancements made to the selected middleware follow the guidelines presented in Section 5.5. Features introduced to Apache Axis2 [Apache Software Foundation, 2009] are presented as an example of how these

guidelines could be used in enhancing a stand-alone web services middleware server. The second case study is an example of how the guidelines will help in achieving predictability of execution in a cluster setup hosting web services. We enhanced Apache Synapse [Apache Software Foundation, 2008], an ESB product to act as the dispatcher of the cluster and use the enhanced version of Axis2 as the executors hosting the web services. The case studies are presented as follows. For each case study, the enhancements that are generic in nature are presented first without any product specific implementation details. These are generic enough to be directly implemented on any web services middleware. It is followed by specialised changes made to each product with specific implementation details. Conceptually the techniques are still applicable to any web services middleware product.

5.6.1 Development Platform and OS

Apache Axis2 and Apache Synapse have been developed using Java as the development platform. Whilst versions of Axis2 are available also in C, the fully featured Java version is preferred by developers. Moreover, Apache Synapse also uses parts of Axis2 in its core. Therefore, the Java versions of Apache Axis2 and Synapse were selected for this implementation. Java is known to be a platform that lacks predictable execution times due to its design features such as the garbage collection mechanism [Wang and Baglodi, 2002]. Conforming with the guideline G1 in Section 5.5, we use Java Real-time Specification (RTSJ) [Oracle Corporation, 2009a] as the supporting development environment. RTSJ introduces several features (not available in standard Java releases) that support predictability of execution required for applications with stringent time requirements. For instance, it introduces several new strictly enforced priority levels that directly map on to proper OS level counterparts. Moreover, it also contains a new real-time thread class that can be empowered with the aforementioned priorities to ensure uninterrupted execution even from the garbage collector mechanism. RTSJ also provides high precision clocks that could be used for timing in such applications upto a nanosecond accuracy.

For RTSJ to function properly, it needs to be deployed upon an OS with real-time features. Conforming with guideline G1, we use Sun Solaris 10 (SunOS) with real-time kernel modules as the underlying OS for the solution. SunOS provides RTSJ with direct mapping onto available priorities and prioritised resource allocations, in order to

maintain the level of predictability required.

5.6.2 Introduction of a Deadline

Predictability of execution is all about ensuring the completion of request execution within a perceived time period. Following guideline G2, a deadline is introduced into each web service invocation. A client of a particular web service hosted on middleware supporting predictability, can decide on a suitable deadline and specify it at service invocation.

While this could be done in multiple ways for both SOAP based and RESTful services, for this implementation we communicate the deadline to the server using SOAP headers. However, it could also be conveyed as part of the payload for RESTful services. By using the SOAP headers, the syntax of the service invocation nor the payload is modified and the deadline which can be considered as metadata is accessed separately from the service parameters.

```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
 <soapenv:Header>
  <edu.rmit.cs.rt:RealTimeParams xmlns:edu.rmit.cs.rt="http://www.RealtimeSOAP.org">
   <edu.rmit.cs.rt:Deadline>
    1500
   </edu.rmit.cs.rt:Deadline>
  </edu.rmit.cs.rt:RealTimeParams>
 </soapenv:Header>
 <soapenv:Body>
  <ns1:calculatePrimesService xmlns:ns1="http://endpoint.testservice">
   <ns1:primeLimit>100000</ns1:primeLimit>
  <ns1:calculatePrimesService>
 </soapenv:Body>
</soapenv:Envelope>
```

Figure 5.5: Sample SOAP message with deadline

Figure 5.5 contains a listing of a sample SOAP message with the newly added deadline information highlighted. The deadline specified in milliseconds, is the ultimate time limit the service invocation is requested to complete within.

5.7 Stand-Alone Implementation

In this section, we present the enhancements made to Axis2, the stand-alone web services middleware product we chose for the implementation. Note that although we present Axis2 for the case study, the enhancements are generic enough to be applied to any other middleware product available.

Apache Axis2 is a highly modular web services middleware that is widely used. Inherently it supports both SOAP based and RESTful web services and has been designed with maximising throughput in mind. It provides a framework to customise the processing of a web service request through *handler* objects, while keeping the core functionality unchanged. The processing of a web service request goes through multiple modules in Axis2. Request execution happens in a *best-effort* manner, through a thread pool where each worker thread is tasked with the complete execution of a request. A similar thread pool with a single worker thread is used as a listener for incoming requests. A web service request is represented within Axis2 using a hierarchical and self contained Information Model which is available to any of the functional modules. Therefore, it also acts as a message, carrying the necessary information throughout each stage of execution.

5.7.1 Schedulability Check Based Admission Control

A major change required to achieve predictability in web services middleware, is the conditional acceptance of requests. Herein, requests must be selected based on their laxity property. Following guideline G3, every request is subjected to a schedulability check that follows Algorithm 1, presented in Chapter 3. However, when incorporating this algorithm into the middleware, care was taken to prevent any issues with concurrency. Algorithm 6 lists out the modified algorithm with the concurrency constructs in place. Algorithm 1, Line 6 contains the calculation of processor demand using Equation 3.5.3. The remaining execution time of an already accepted request is defined in Equation 5 as the difference between its execution time requirement and the time already spent in executions. In its actual implementation a two pronged approach was taken. Given the unknown nature of request properties, it is impossible to know the execution time requirement of a request. However, this can be facilitated by keeping execution time history information. It can be kept as an average for the operation, for

Algorithm 6 Schedulability check algorithm with concurrency constructs

Require: New request N, Queue of Accepted Requests RQ
Ensure: N is accepted or rejected

1. Enter Critical Section
2. PDW $\leftarrow$ 0; PDA $\leftarrow$ 0; withinTasksChecked $\leftarrow$ $false$
3. withinTasksChecked $\leftarrow$ $false$
4. RQ.acquire
5. **while** RQ has more **and** withinTasksChecked is $false$ **do**
6. nextReq $\leftarrow$ RQ.getNextReq
7. **if** nextReq.startTime $\geq$ N.startTime **and** nextRequest.deadline $\leq$ N.deadline **then**
8. **if** Exec. Info. for nextReq.Operation exists **then**
9. PDW $\leftarrow$ PDW + nextReq.getRemainingTime
10. **else**
11. PDW $\leftarrow$ PDW + getGlobalAverageExecTime
12. **end if**
13. **else**
14. **if** nextReq.deadline $\geq$ N.deadline **then**
15. withinTasksChecked $\leftarrow$ $true$
16. **end if**
17. **end if**
18. **end while**
19. **if** Exec. Info. for N.Operation exists **then**
20. PDW $\leftarrow$ PDW + N.getRemainingTime
21. **else**
22. PDW $\leftarrow$ PDW + getGlobalAverageExecTime
23. **end if**
24. LoadingFactor $\leftarrow$ $\frac{PDW}{N.deadline - N.startTime}$
25. **if** LoadingFactor > 1 **then**
26. RQ.release
27. **return** $false$
28. **end if**
29. PDA $\leftarrow$ PDW
30. **while** RQ has more requests **do**
31. nextReq $\leftarrow$ RQ.getNextReq
32. **if** Exec. Info. for nextReq.Operation exists **then**
33. PDA $\leftarrow$ PDA + nextReq.getRemainingTime
34. **else**
35. PDA $\leftarrow$ PDA + getGlobalAverageExTime
36. **end if**
37. LoadingFactor $\leftarrow$ $\frac{PDA}{nextReq.deadline - N.startTime}$
38. **if** LoadingFactor > 1 **then**
39. RQ.release
40. **return** $false$
41. **end if**
42. **end while**
43. RQ.insert(N)
44. RQ.release
45. **return** $true$
46. Exit Critical Section

the combination of its input parameters. Similarly, a global average can also be kept for the service. Both averages could be updated at the end of a service invocation. When history information exists for a particular operation, the average for the set of inputs can be used in the calculation (Algorithm 6 : Lines 9, 20 and 33). In the case of using the set of input for the first time, the global average could be used instead (Algorithm 6 : Lines 11, 22 and 35).

In implementing the schedulability check, efficiency is further achieved by using an ordered queue (RQ) for accepted requests which automatically inserts a request to the proper position in the queue based on its deadline. This process prevents the sorting of requests happening on each execution of the algorithm thereby reducing the time complexity to *O(n)* amortized. Given time complexity of Algorithm 1 presented in Chapter 3 to be also *O(n)* and the additional steps in Algorithm 6 not having a different effect on its execution, the complexity analysis presented in Chapter 3 remains valid for Algorithm 6 as well.

Recall from chapter 3 that the schedulability check works by considering the laxities of requests already accepted at the server. Therefore, the acceptance of a request will be decided on how compatible its laxity is with already accepted requests. If the acceptance of new requests by the middleware takes place while the schedulability check is carried out for others, it results in race conditions. Preventing such phenomenon the entire schedulability check is marked as a critical section (Lines 1 and 44). Moreover, the list of accepted requests (RQ) would be modified when a request completes execution or when a request is accepted for execution. These events would also turn into race conditions if access to the list is not controlled. Concurrent access to the list is controlled through the use of a binary semaphore and the schedulability check secures a lock on it prior to it being read (Line 4). In the event of accepting a request for execution the lock is released (Line 44) after queueing the request for execution (Line 43). If the request is not schedulable the lock is released prior to exiting from the algorithm (Lines 26 and 39).

5.7.2 Priority Model

Typical web services middleware contain no mechanisms to differentiate request processing. Therefore, all requests are executed at the same priority level. Guideline G4, mandates fine-grain differentiation in request processing, for achieving execution time

predictability. Axis2 by default, does not use different priority levels in processing requests. Following this guideline, we introduce three priority levels to achieve differentiation in the functionality of the middleware.

Priority	*Purpose*	*Mapping*
Lowest	Used for metadata exchange such as WSDL or Schema requests	Set to the highest priority available on standard Java. Can be interrupted by the GC
Mid-Level	Execution prevention priority. Used on worker threads to preempt and suspend them from execution. All threads assigned with a request but currently not in execution will have this priority assigned to them	Set to the mid level of the RTSJ priorities. Can be interrupted by the GC
Highest	Execution granting priority. Used on a worker thread to grant the CPU for execution. At most, only one thread per execution lane is assigned with this priority	Set to the highest priority level available on RTSJ. Cannot be interrupted by the GC

Table 5.1: Priority Levels Introduced

The extended priority levels available in RTSJ provides a better mapping of thread level priorities to OS level priorities. Moreover, the *Highest* priority level introduced to the system is guaranteed to be uninterrupted by the GC as well as any process outside the Java Virtual Machine. These priority levels are used by a newly introduced real-time scheduler component at runtime, to achieve fine-grain differentiation in request processing. By using these priority levels on worker threads, the real-time scheduler is able to control their execution and the order of completion of requests.

5.7.3 Real-time Scheduler and Thread Pools

The discussed priority model is used by a real-time scheduler component newly introduced to web services middleware. The scheduler ensures the ordered execution of requests based on a pre-defined scheduling algorithm. The algorithm used for scheduling can be configured and for the proposed solution, EDF scheduling is implemented, following guideline G2. As the execution in web services middleware is carried out by one or more thread pools, the introduced real-time scheduler is designed to use a custom

made real-time thread pool to manage execution. The scheduler manages execution by enforcing the priority model on worker threads in the pool.

EDF Based Scheduling

The scheduler uses Algorithm 7 to reschedule the execution of threads upon the assigning of a new request (N) for execution. All threads with requests currently in execution, are kept track of using a list of references (LT) by the real-time scheduler. The number of requests executing concurrently can be configured and is usually decided on the number of processors available on the server. The worker thread assigned with the request having the earliest deadline at a given time, would be in execution while the others will be queued on TQ, waiting for their turn to re-claim the CPU in the order of their deadlines. The deadline of N is compared sequentially with requests referenced by the members of LT (Lines 4-21). If any of the references do not have a request already assigned for execution, the new request is assigned to it immediately and the priority of the worker thread is set to *High*, for it to claim the processor (Lines 5-7). If all references have assigned requests, the deadline of N is compared with each of them (Line 12). If N has an earlier deadline than any one of them, the worker thread with the latest deadline is preempted by setting the priority to *Mid* (Line 13) and subsequently it is queued in TQ for resumption later (Line 14). Thereafter, the reference is set to the worker thread of N and it is allowed to claim the processor by increasing its priority to *High* (Lines 15,17). However, if the deadline of N is later than that of all requests currently in execution, N is prevented from further execution and is queued for resumption later (Lines 23-26). While the rescheduling takes place and tasks with the earliest deadlines are selected, the number of threads must remain unchanged. Access to LT is controlled using a semaphore, to prevent any changes while a scheduling run is in process. The algorithm acquires a lock on the object (Line 3) and releases it as soon as the operations are completed (Line 22). Furthermore, the entire Algorithm 7 is marked as a critical section (Lines 1 and 27), to prevent any race conditions.

Given the number of processors on the server, there could be a request in execution at each one of them. However, the requests that are being executed are guaranteed to be the ones with the earliest deadlines of all requests accepted. When a newly accepted request needs to be executed immediately due to an earlier deadline, the request preempted must be the one with the latest deadline out of the ones executing (it may not be the one N

Algorithm 7 EDF Implementation - Scheduling of Threads

Require: Thread Queue TQ, Ordered active thread pointer list LT, New request N
Ensure: Execution of Threads assigned with earliest deadlines

```
1.  Enter Critical Section
2.  found ← false
3.  LT.acquire
4.  while found is false and LT.hasMore do
5.     if LT.ptrNextThread is not assigned then
6.        LT.ptrNextThread ← N.getThread
7.        N.getThread.priority ← High
8.        LT.resetLatestThread
9.        found ← true
10.    else
11.       R ← LT.ptrNextThread.getRequest
12.       if N.deadline < R.deadline then
13.          LT.ptrLastThread.priority ← Mid
14.          TQ.queue(LT.ptrLastThread)
15.          LT.ptrLastThread ← N.getThread
16.          LT.resetLatestThread
17.          LT.ptrLastThread.priority ← High
18.          found ← true
19.       end if
20.    end if
21. end while
22. LT.release
23. if found is false then
24.    N.getThread.priority ← Mid
25.    TQ.queue(N)
26. end if
27. Exit Critical Section
```

was last compared with). To make this selection efficient, the scheduler keeps a special pointer (*ptrLastThread*) directly referencing the thread with the latest deadline, out of all that is in execution. This ensures a quick preemption between N and the target request with the latest deadline. Once the preemption is complete the *ptrLastThread* needs to be reset to reference the worker thread with the latest deadline. In Algorithm 7, this step is carried out after N starts execution (Lines 8 and 16). Algorithm 8 carries this out by a sequential comparison of deadlines. At the end of the comparison process, *ptrLastThread* contains a reference to the thread with the latest deadline.

Algorithm 8 LT.resetLatestThread Implementation

Require: Ordered active thread pointer list LT
Ensure: ptrLastThread points at the thread having the request with the latest deadline

```
1. if LT is not empty then
2.    ptrLastThread ← LT.first
3.    for all thread ∈ LT do
4.       req ← thread.getRequest
5.       if req.deadline > ptrLastThread.deadline then
6.          ptrLastThread ← thread
7.       end if
8.    end for
9. end if
```

Complexity Analysis of EDF Algorithms

As Algorithm 7 uses Algorithm 8 in its functionality, we first look at the time complexity of Algorithm 8.

Let n be the number of active thread pointers in list LT. Let $T(n)$ be the running time of the algorithm. LT allows constant time access to the first element in the list irrespective of its length. It can be observed in line 3 of the algorithm that all members of the list LT would be accessed individually. Therefore, lines 3-8 will execute n number of times. Consider time taken for execution of line 2 is c_1, to access the current element in LT is c_{LT}, for execution of line 4 is c_2 and for execution of lines 5-7 is c_3 at maximum. Therefore, running time of Algorithm 8 can be calculated as,

$$T(n) = c_1 + nc_{LT} + nc_2 + nc_3$$

Then, $c_1 + nc_{LT} + nc_2 + nc_3 \leq n(c_1 + c_{LT} + c_2 + c_3)$ for all $n > 1$. Therefore it can be concluded that T(n) is in O(n) in the worst case. Care must be taken in implementation to use a technique such as using the Iterator pattern [Gamma et al., 1995], that allows constant time access to each element in LT.

Next we analyse the time complexity of Algorithm 7. Let n be the number of active thread pointers in list LT. Let $P(n)$ be the running time of the algorithm. Statement in line 4 repeats the statements within lines 5-20, until *found* is set to *true* or until all members of LT has been accessed. The best case scenario is that none of the thread pointers (accessed through *LT.ptrNextThread*) in LT has threads assigned. In which case the condition in line 5 results in being true and lines 5-20 is executed just once. On

the contrary, the worst case is when either the last thread pointer in RT is not assigned or when the new request has a deadline later than all requests currently in execution. In both instances lines 5-20 are repeated n times. However, in the earlier case lines 6-9 gets executed and in the latter lines 11-18 gets executed. Moreover, lines 24 and 25 also executes in the second instance.

With the previous analysis it was concluded that the operation *LT.resetLatestThread* detailed by Algorithm 8 is in O(n) in the worst case. Although this operation is used in lines 8 and 16, they will get executed at most once. Therefore, within lines 4-21, the only statements that gets executed n times would be lines 5,11 and 12. Let t_1 be the time taken for execution of lines 5,11 and 12. Let t_2 be the total execution time of lines 6,7 and 9. Let t_3 be the total execution time of lines 13-15 and 17-18. Let t_4 be the total execution time of lines 23-26. The running time of the algorithm can be calculated as,

$$
\begin{aligned}
P(n) &= nt_1 + t_2 + T(n) \mid nt_1 + t_3 + t_4 + T(n) \\
P(n) &= nt_1 + t_2 + n(c_{LT} + c_2 + c_3) + c_1 \mid nt_1 + t_3 + t_4 + n(c_{LT} + c_2 + c_3) + c_1 \\
P(n) &= nt_1 + t_2 + n(c_{LT} + c_2 + c_3) + c_1 \leq n(t_1 + t_2 + c_{LT} + c_2 + c_3 + c_1) \mid \\
&\quad nt_1 + t_3 + t_4 + n(c_{LT} + c_2 + c_3) + c_1 \leq n(t_1 + t_3 + t_4 + c_{LT} + c_2 + c_3 + c_1)
\end{aligned}
$$

With the above, it can be concluded that P(n) is in $O(n)$ in the worst case. Similarly, for the best case scenario, the execution passes through the algorithm only once and all pointers in LT contains empty references. Therefore, it can be concluded that P(n) is in $\Omega(1)$ in the best case.

Real-time Thread Pool Design

Figure 5.6 illustrates the design of the real-time thread pool and the real-time scheduler component. The thread pool, which is an instance of the *RTThreadPoolExecutor* class contains worker threads which are objects of the *RTWorkerThread* class that inherits from the RTSJ *RealtimeThread* class. *RTThreadPoolExecutor* also contains an instance of the scheduling algorithm *RTScheduling*, used for scheduling the execution of worker threads. For the case study presented, we used an instance of *RTEDFScheduler*. The scheduler controls the execution of worker threads through the *RTThreadPoolExecutor*,

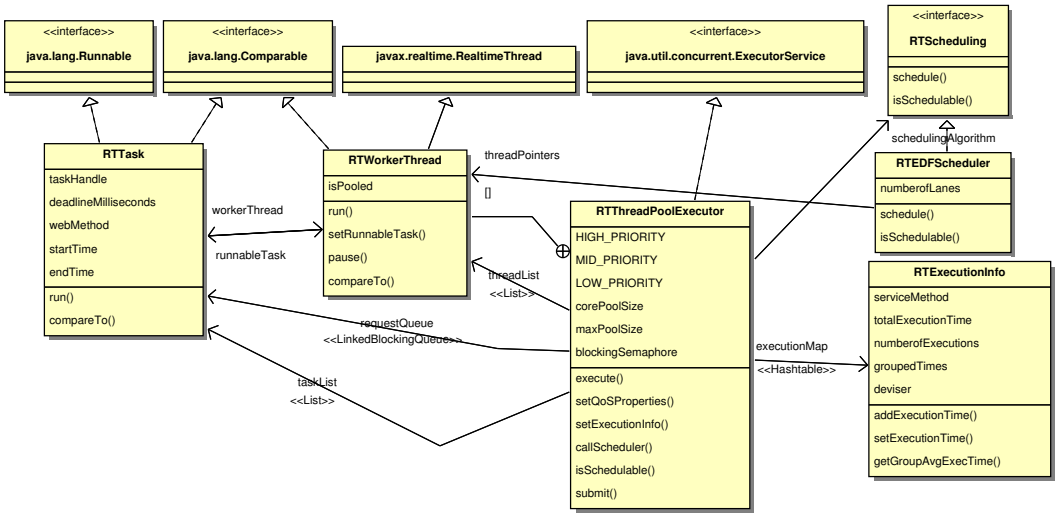

Figure 5.6: Real-time Thread Pool Class Diagram

by enforcing the aforementioned priority model. Requests are internally represented as instances of *RTTask*, which are assigned to a *RTWorkerThread* on creation. *RTExecutionInfo* instances store summarised execution time history that is used for the schedulability checks done by the scheduler. The level of summaries can be configured and *RTExecutionInfo* instances are stored in a hashtable allowing constant time access and storage. The *RTThreadPoolExecutor* uses *threadList* to keep track of all worker threads and *taskList* to keep track of all request (represented by *RTTask*) instances in the system. Requests are handed over to the threads using a blocking queue (*requestQueue*). All three of these data structures are made thread safe, and accessing them is managed using concurrency constructs.

Integration into Axis2 Functionality

Figure 5.7 summarises the specialised enhancements made to Axis2. Both thread pools were replaced with real-time thread pools. To take advantage of multi-core / multi-processor hardware, the executor thread pool was configured to have *n-1* execution lanes (where *n* is the number of cores / processors on the server). Therefore, at a given time the requests with the *n-1* earliest deadlines will get executed. This number was decided based on the optimality principles discussed in [Subramaniam V., 2011] and the recommendations made in the RTSJ documentation [Oracle Corporation, 2009a]. Moreover, this allows other active components such as the listener to operate freely without being interrupted by higher priority worker threads used for execution.

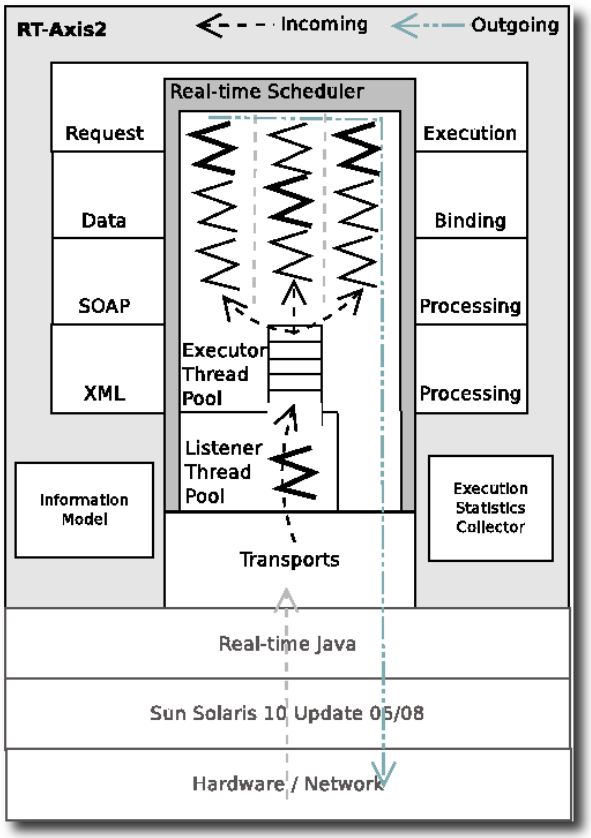

Figure 5.7: RT-Axis2

For illustration purposes, Figure 5.7 contains 3 execution lanes, each with the currently active thread highlighted. Both thread pools were set to pre-create the worker threads at system start-up to avoid the overhead in thread creation. The functionality of the thread pools are managed by the newly introduced real-time scheduler component. As observed, it manages the execution of all worker threads across all functional modules within the enhanced version of Axis2 (RT-Axis2), using an EDF based scheduling algorithm. The sequence of events inside RT-Axis2 when a request is received, until the completion of its execution is presented in two sequence diagrams. Figure 5.8 summarises the events that take place in the scheduling phase and Figure 5.9 the post scheduling phase.

As mentioned previously, the deadline for each service invocation is conveyed to the server using SOAP headers. Thus, extracting this information was done by implementing additional functionality in the XML processing module. Upon extraction, this information is stored and passed through to other modules with use of an extended Axis2

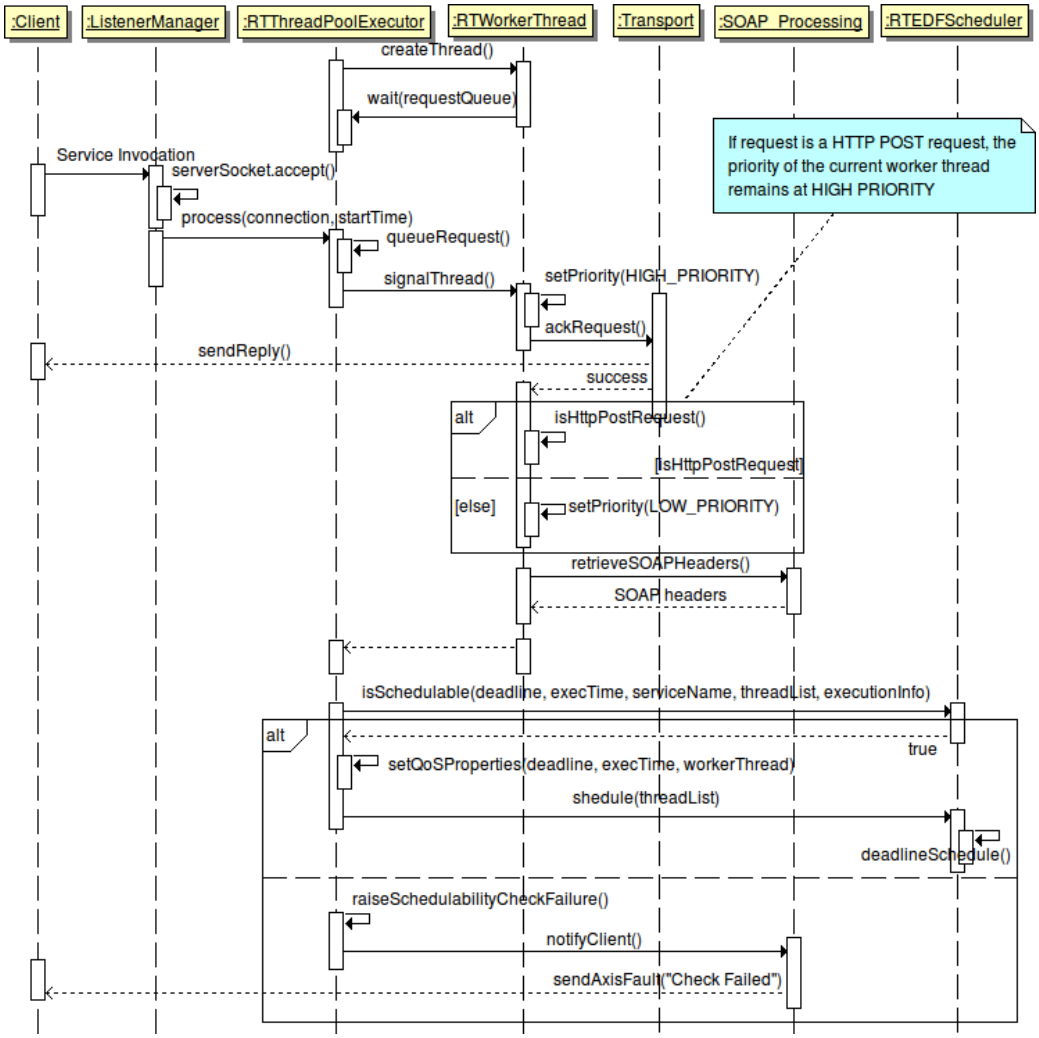

Figure 5.8: RT-Axis2 Execution - Sequence of Events - Scheduling Phase

Information Model. When the execution continues onto the SOAP processing module, identification of the request is done and any metadata requests such as for WSDL documents would have the real-time scheduler demote the worker thread to a *Low* priority. If the request is identified to be a service invocation, the schedulability check is carried out immediately using the deadline information now available in the information model. Furthermore, execution time history information available through a newly introduced Execution Statistics Collector module is internally used by the real-time scheduler to conduct the check. The schedulability check takes place within the SOAP processing module. If the check fails, further processing of the request is suspended immediately and the client is notified using the already built-in fault mechanism of Axis2. Conversely, if the new request can be scheduled, the scheduler immediately conducts

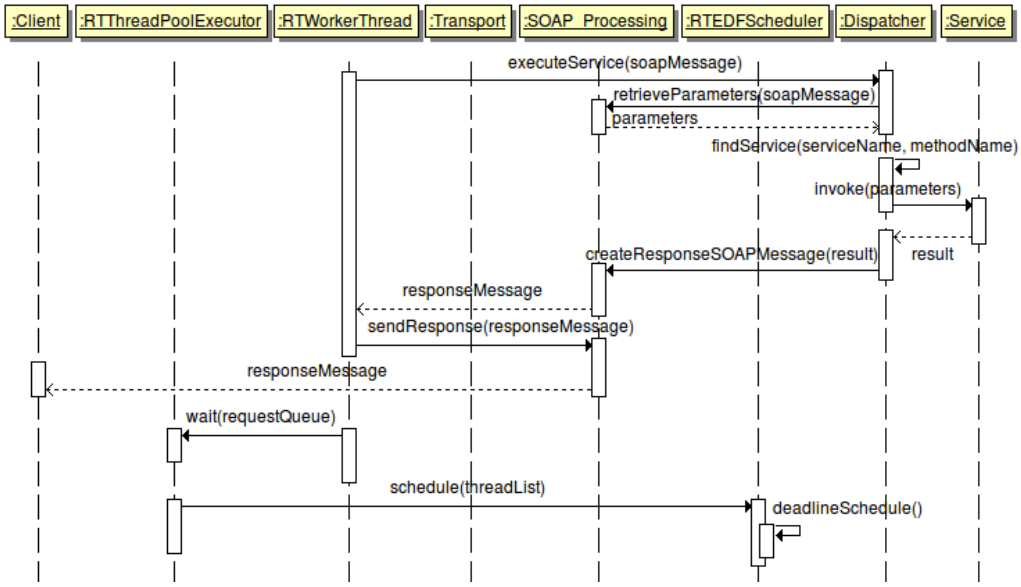

Figure 5.9: RT-Axis2 Execution - Sequence of Events - Post Scheduling Phase

a rescheduling of threads, upon which at most only the execution of a single request out of all active will be interrupted (following aforementioned Algorithm 8). The execution continues onto a normal service invocation process where the results would be conveyed back to the client. Once entire processing of the request is completed, housekeeping activities such as updating execution time history records with times obtained from the current invocation, takes place in the Statistics Collector Module, prior to the worker thread returning back to the pool. Once a worker thread completes the assigned request, the scheduler would re-assign it with the next request at the head of the queue for execution.

While it is possible to influence the request processing in Axis2 through the *handler* framework rather than making any modifications to the modules themselves, doing so does not allow complete control over the execution in core modules. As handlers sit outside the core of Axis2, they have no means of influencing the internal fine-grain execution of requests. Moreover, the overhead created by the enhancements required for predictability was kept to a minimum by changing the core modules themselves, which enabled decision making (i.e. schedulability check) as soon as the required information is available. Therefore, the required enhancements were made to the core of Axis2.

5.8 Cluster-Based Implementation

In this section, we present the implementation details of the cluster setup, hosting web services. A cluster constitutes of a dispatcher and a set of servers that host the services. With the single server implementation, request processing and execution happened on the same host. Herein, we refer to all the necessary processing prior to being ready for service invocation as request processing. This includes but not limited to, obtaining the SOAP or XML message structure, obtaining the SOAP headers and service parameters and conducting the schedulability check on a new request. One of the main goals in the cluster implementation is to free the servers as much as possible of the request processing overhead, and allocate more processing resources to request execution.

5.8.1 Implementation Choices

A cluster can be implemented in many ways. Here we consider three possible implementations, to choose the one with greatest ability for achieving predictability of execution. Figure 5.10 illustrates the three cluster models described below.

A. Dispatching is done in a content-blind manner without the use of any content-based dispatching algorithm. Request processing and execution both happens at a cluster server. Direct communication between cluster servers and client.

B. Content-aware or content-blind request dispatching. Dispatching is transparent to the client and subsequent communication happens between a cluster server and a client directly. Request processing happens both at the dispatcher and the cluster server. However, schedulability check is conducted by each server individually.

C. Dispatching decisions are made only based on the request content. Dispatching is transparent to the client and all communication happens between dispatcher and cluster server directly. Schedulability check happens only at the dispatcher and so does the majority of the request processing. The request processing happening at each cluster server is just to fetch the request assigned by the dispatcher.

Cluster Model A is the simplest cluster implementation with dispatching decisions being content-blind. As in the case of the stand-alone implementation, the servers hosting the web services content performs both request processing and request execution. In fact,

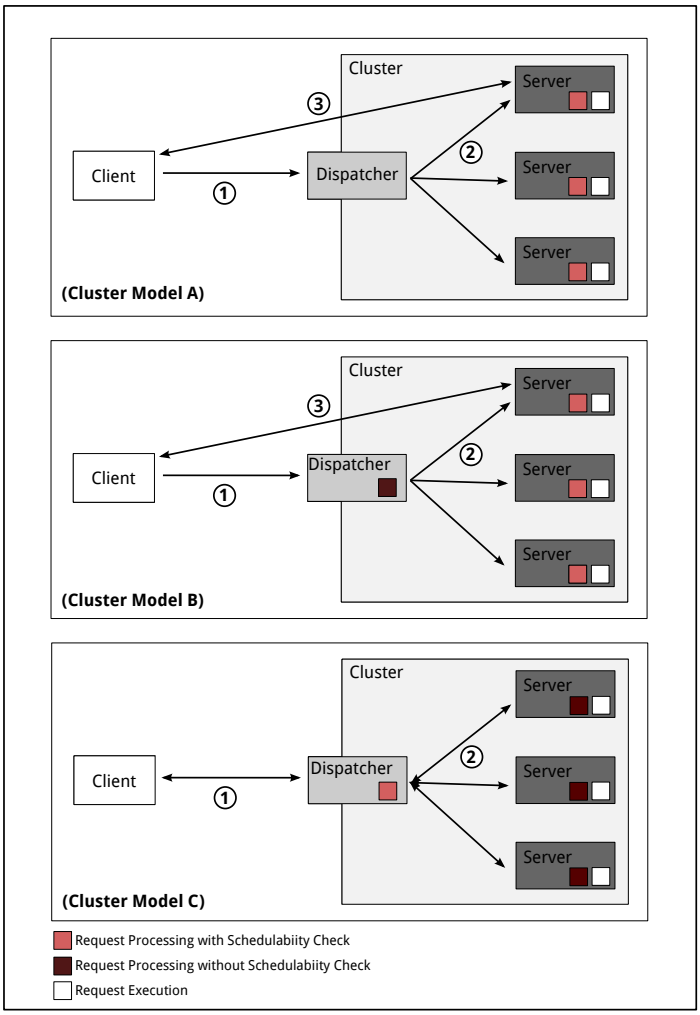

Figure 5.10: Cluster Implementation Models

the enhanced version of Axis2 could be used for the servers with a simple load balancer or an ESB with request routing capabilities acting as the dispatcher, to implement this model. While the advantage of this model is its simplicity to implement, there are many disadvantages that makes it the least preferred for our implementation. Firstly, there is no separation of request processing and schedulability check, from service execution. The servers hosting the services are the servers hosting the services are not free from the overhead of the request processing. While the time and resources spent on this maybe relatively lower compared to the invocation of a CPU bound service, it still demands of processing resources and will share the processor with the service invocations, making it an overhead. Secondly, content-blind dispatching does not consider the priority of a request as a parameter in making dispatching decisions. Lastly, once the request is dis-

patched to a server, direct communication takes place between the server and the client, making the cluster server visible to the client. Therefore, a client could directly send subsequent requests to the particular server by passing the dispatcher. Nevertheless, this model will result in an increased level of schedulability compared to the single server implementation, as the requests are distributed multiple servers.

Cluster Model B contains a dispatcher which could dispatch requests either in content-aware or content-blind manner. When the latter is used, the cluster performs identically to *Cluster Model A*. If the dispatching decisions made are content-aware, request processing at the dispatcher ensures the necessary parameters are available to make the dispatching decisions. Once a request is assigned to a server, subsequent communication between the server and the client happens directly. Therefore, the dispatcher does not keep any information about the request dispatched to the server. As a result, the request processing at the dispatcher does not involve the schedulability check. As in the case of a single server implementation, each server in the cluster is required to carry out request processing that includes the schedulability check. Acceptance or rejection of a request is conveyed directly to the client and the dispatcher has no knowledge of the result. Thus, the enhanced version of Axis2 used in the single server implementation can be directly used as cluster servers in this implementation. This model enables the dispatching decisions to be content-aware and be made using a property of a request such as the deadline or laxity. However, the direct communication with clients prevents the server from keeping any state about the requests. As the servers are required to conduct the schedulability check, request processing cannot be isolated from request execution, as intended. Similar to Model A, the servers are directly visible to the clients. Clients could bypass the dispatcher and send subsequent requests directly to the server, not being subject to some of the predictability features of the cluster

In *Cluster Model C*, the cluster is visible to the clients as a single system. All communications with the cluster happens through the dispatcher. The dispatcher functions in a content-aware manner. The request processing that takes place in the dispatcher includes the additional step of the schedulability check. The intended order of functionality is that a request is first mapped onto a server using the dispatching algorithm. Thereafter the request is checked for schedulability on the selected server. The request is physically dispatched to the assigned server only upon the success of the schedulability check. For the dispatcher to conduct the check for every server in the cluster, it must have up-to-date information about the requests at each server. This is made possible by

the design where the dispatcher is the central node of communication in and out of the cluster. Upon the completion of a service invocation, the results are sent back to the client through the dispatcher. This enables the dispatcher to update its records about the request just completed. Moreover, this also enables the dispatcher to queue the requests assigned to an executor until the server is ready for execution. These steps free the servers from the overhead of the schedulability check, most of the request processing and enables them to use all their processing resources for service execution. The request processing that happens at each server consumes much less resources compared to the activities at the dispatcher. Moreover, this would happen only when a request is dispatched to the server, when it is to be executed. In a predictability standpoint it is a further guarantee on uninterrupted service execution, which increases the chances of meeting a requested deadline.

Due to these reasons, *Cluster Model C* is chosen as one that would achieve the best predictability results. The cluster is implemented using a slightly modified version of the enhanced Axis2 used for the single server implementation, as cluster servers and an enhanced version of Apache Synapse as the dispatcher. While the enhancements made are for these specific products, conceptually they could be applied for any cluster setup hosting web services. The concepts are most suitable for a locally distributed cluster as the dispatcher is the central point of communication with clients.

5.8.2 Executor Implementation

While Synapse has the ability to support any server hosting web services as executors in a cluster, we use Axis2 due to our familiarity with the product. Moreover, as the executors need to ensure predictability of execution, the enhancements presented in the stand-alone implementation are directly applicable here with minor modifications. Recall that in a stand-alone web services middleware, request processing, admission control and request execution all happen within the same middleware instance. This arrangement at times leads to request execution being interrupted by processing and admission control of other requests.

With the chosen cluster-based implementation, admission control takes place at the dispatcher. As a result, request processing is required to take place at the dispatcher, prior to the admission control mechanism. This allows the executors to be relieved from admission control and request processing tasks, to be dedicated request executors. To

minimise the interruption to service execution, requests will only arrive at the executor when it is scheduled to be executed. Until such time, requests ready for execution are queued at the dispatcher separately for each executor in the cluster. Interruptions to service execution at an executor would only happen when the dispatcher assigns a request with an earlier deadline than that to the on in execution.

In preparing RT-Axis2 to act just as an executor, the schedulability check is removed from the request processing. Requests directed to it are directly accepted. If a request is in execution when the server receives a request from the dispatcher, the new request will always be having an earlier deadline. After confirmation by the real-time scheduler, the request currently in execution will be preempted and kept in a queue for resumption later. The scheduler allows the new request to gain the processor for execution. Another modification done to the server is for the server to check the preempted queue for requests, at the completion of an invocation. Upon finding requests in the queue the real-time scheduler will resume their execution in the increasing order of their deadlines. Moreover,the result of a service invocation is also conveyed back to the dispatcher.

5.8.3 Dispatcher Implementation

The dispatcher functionality is implemented in the cluster setup, using Apache Synapse. Synapse is a lightweight ESB product widely used for enterprise integration in service oriented computing. Designed for message mediation, it is optimised for throughput and processes requests in a *best-effort* manner. ESBs differ from the typical web services middleware where services are hosted, as they mainly function as message exchanges and gateways where transformations between multiple protocols are supported. The architecture of Synapse is based on Axis2 and contains the Axis2 engine in its core. In processing messages, services such as XML processing and SOAP processing are facilitated by the Axis2 core. Similar to the design of Axis2, Synapse has an extensible architecture. In and out flows of messages through Synapse could be influenced by programmers using this framework which are modelled as *Sequence* and *Endpoint* objects. Due to such message mediation features, Synapse was an ideal candidate as a dispatcher in a cluster hosting web services.

Synapse employs several thread pools for its operations. As illustrated in Figure 5.11, all thread pools were replaced with our real-time thread pools presented earlier as part of the enhancements. All real-time thread pools pre-create worker threads to avoid

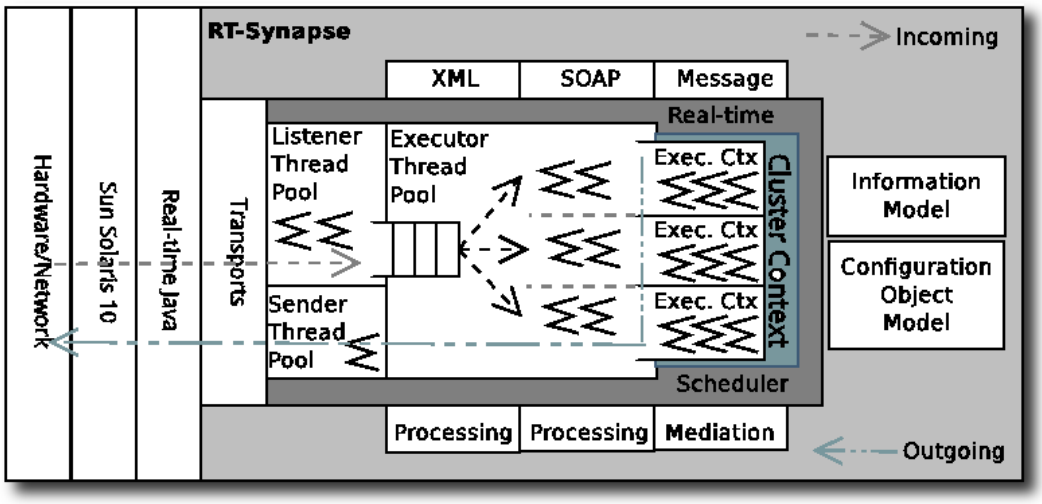

Figure 5.11: RT-Synapse

any delays in object creation. The listener thread pool and the executor pool were configured with *n-1* execution lanes where *n* is the number of cores/processors within the server. The sender pool is configured to have a single worker thread, which is the default in Synapse. Replacing the Axis2 core used by Synapse, with a RT-Axis2 core automatically enables Synapse to have the capabilities such as access to deadline information conveyed through SOAP headers, extraction of deadline information in the XML processing modules and differentiation of request types. A newly introduced real-time scheduler component manages the execution of all worker threads throughout the lifetime of a request inside Synapse. Synapse uses the same information model used in Axis2 and as a result it was easily replaced with the modified version used in RT-Axis2.

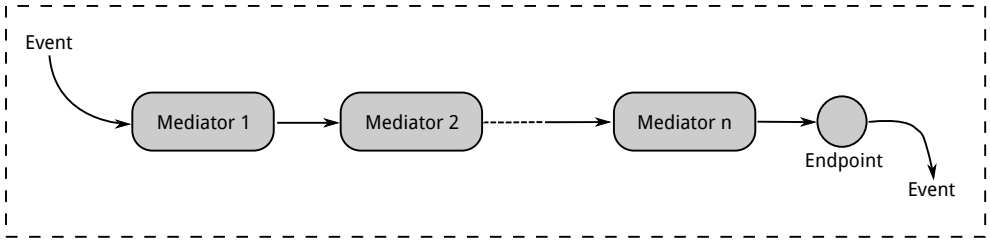

Figure 5.12: Synapse In-Sequence

Mediation of messages in Synapse is carried out using an extensible event driven framework which allows programmers to customise the order and type of mediators used (following the chain of responsibility design pattern [Gamma et al., 1995]). This medi-

ator sequencing starts with an event, goes through an arbitrary number of mediators and ends with an endpoint which results in another event. As seen on Figure 5.12, an incoming message is sent through a sequence of mediators (*in-sequence*), chained together, before being dispatched to the URL given by an endpoint in the sequence. Following this design pattern, we implemented a custom mediator (*RT-LoadBalance Endpoint*) and a sequence (Figure 5.13), that can be configured to use one of the aforementioned dispatching algorithms. It makes use of a standard Synapse *Addressing Endpoint* to dispatch the request to the URL of the executor at the completion of the sequence.

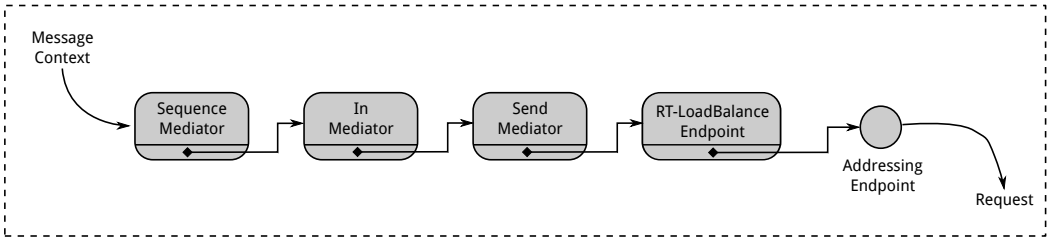

Figure 5.13: RT-Synapse In Sequence

In the stand-alone implementation, the server did a multitude of tasks prior to the actual service invocation. An important design decision made for our cluster implementation was to keep the interruptions to the processing that happens at an executor, to a minimum. By design, all the necessary pre-invocation processing (such as schedulability checks and rescheduling of request execution on arrival of a new request) is done at the dispatcher. Thus, the state at each executor and the overall cluster is kept track of at the dispatcher. As illustrated in Figure 5.14, state of an executor is stored in an *ExecutorContext* instance. State of the overall cluster is kept in a *ClusterContext* instance. Three ordered queues (based on increasing deadlines) are used to queue requests assigned to an executor. Requests are represented in the system as RTTask instances. Requests waiting to be executed are queued in *WaitingQueue* within the *ExecutorContext*. Once a request is dispatched for execution, its representation is queued in a *SubmittedQueue*. Finally, requests that are preempted from execution are queued in a *PreemptedQueue* at the executor.

Any accepted requests with deadlines later than that of the request currently at the executor, are queued in *WaitingQueue* until their turn for execution. At the completion of a service invocation at the executor, the result is returned to the client via the dispatcher,

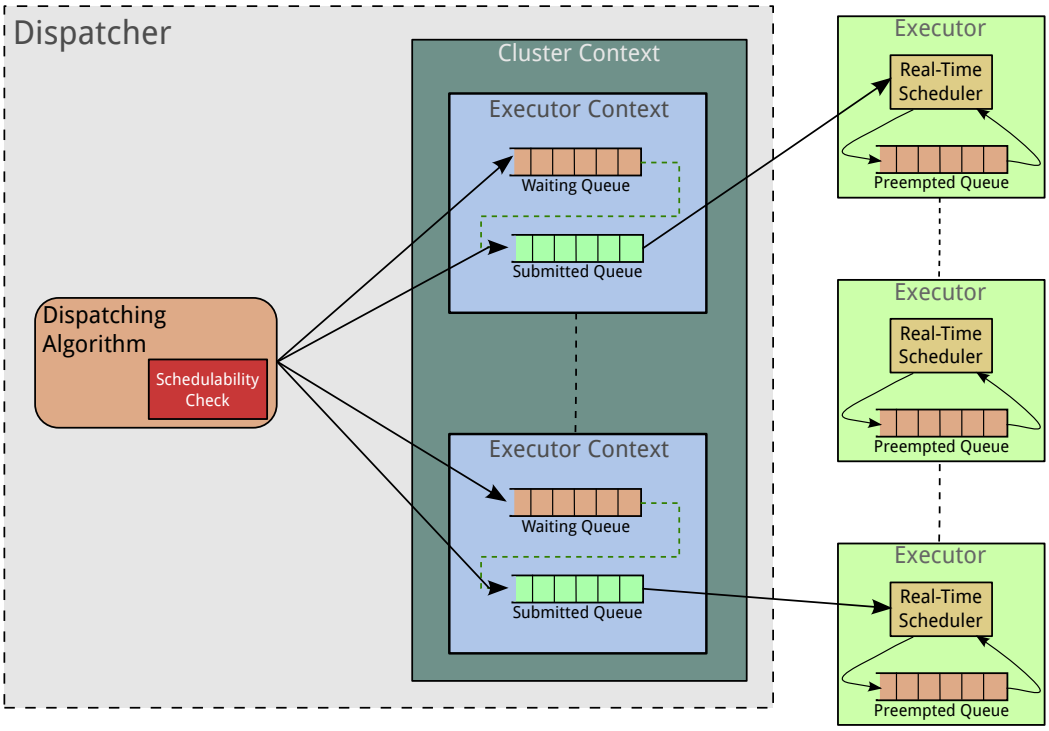

Figure 5.14: RT-Synapse Internals

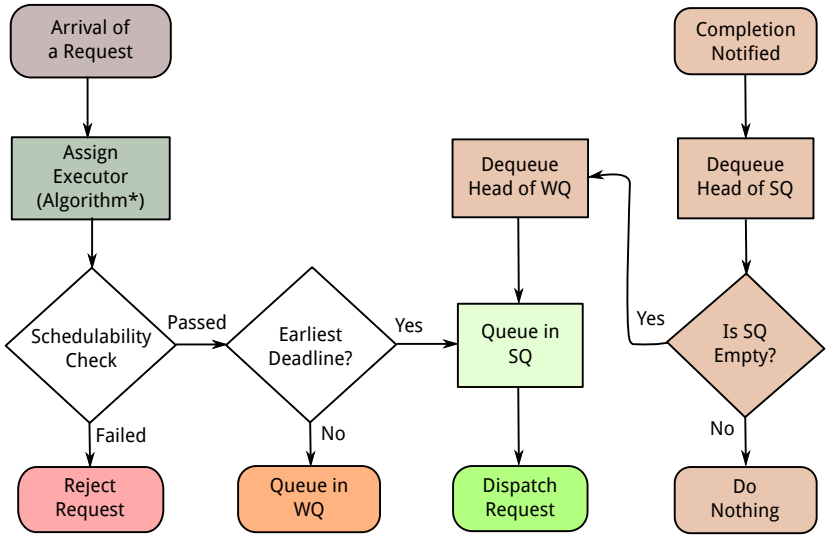

Figure 5.15: RT-Synapse Functionality

where the head of the *SubmittedQueue* will be removed at the same time. Moreover, the execution of any requests waiting in the *PreemptedQueue* is resumed and completed in the order of their deadline. Similarly, if the *SubmittedQueue* becomes empty at the completion of a request, the request at the head of the *WaitingQueue* is removed and

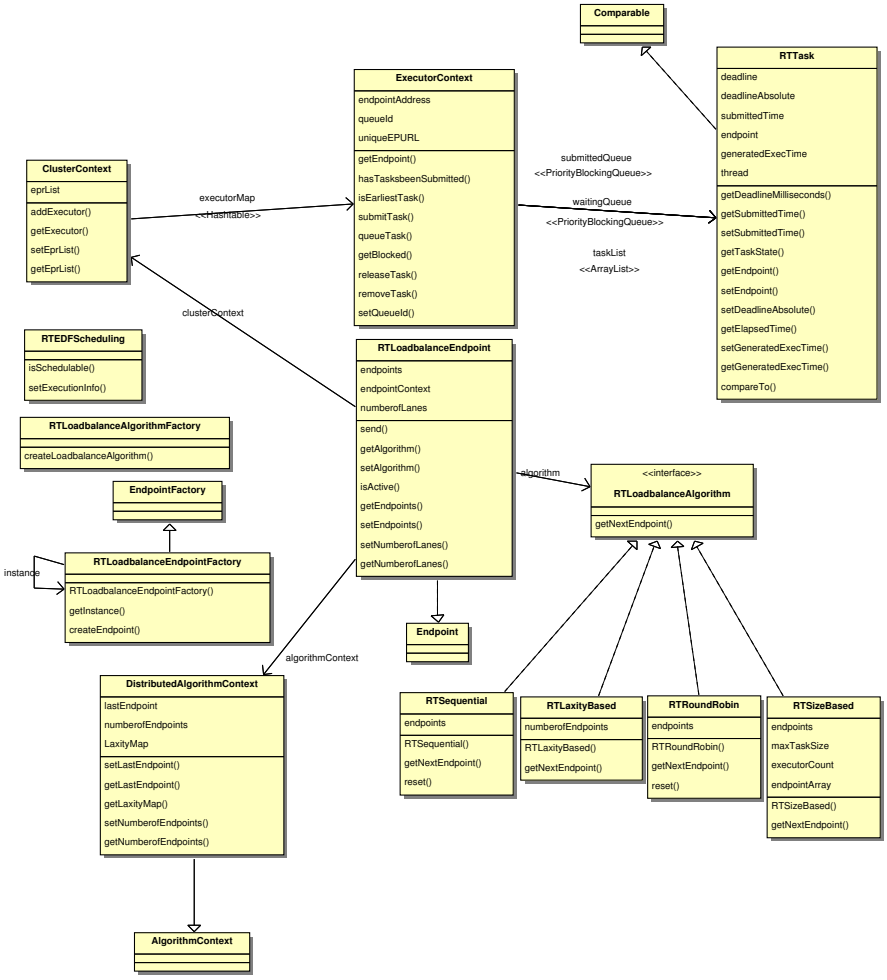

Figure 5.16: Real-time Cluster Class Diagram

dispatched for execution, while its representation (the RTTask instance) is queued in *SubmittedQueue*. Using this design ensures, the processing of a request at an executor is only interrupted by the acceptance of a request with an earlier deadline. Figure 5.15 summaries the sequence of events when a request is dispatched.

Figure 5.16 contains the classes used in the cluster implementation. Requests are represented within the system as *RTTask* instances. By implementing the *java.lang.Comparable* interface, requests can be naturally ordered based on the increasing order of their deadlines by the system automatically. This enables the use of priority queues where requests are automatically sorted in their natural ordering at insertion. Using such self organising priority queues in the implementation results in efficiency in the schedulability check and in dispatching requests. Overall cluster level information is represented in an instance of *ClusterContext* which would have several *ExecutorContext* instances

equal to the number of servers in the cluster, contained within it. Each *ExecutorContext* instance contains a *waitingQueue* and a *submittedQueue*, two priority queues used to keep accepted *RTTask* instances ready for execution and already in execution at the server, respectively.

RTLoadBalancingEndpoint refers to the endpoint implementation that acts as a custom mediator in the sequence. It contains the references to a *ClusterContext* instance and an *RTLoadBalanceAlgorithm* instance which stores dispatching algorithm used by the cluster. State information used y the dispatching algorithm is stored in *DistributedAlgorithmContext* instance. Supplementary classes act as factories in creating the hierarchy of objects used.

5.8.4 Minimising Priority Inversions

Although the techniques we used to minimise priority inversions are product independent, we chose to discuss them separately as they are common to both products used. Priority inversions could impact the execution of a request in two ways. Firstly. a scenario leading to a priority inversion would naturally incur an unwarranted delay in the execution of a request. Following G5, activities that may lead to priority inversion scenarios such as on-screen reporting of operation statuses, recording of output into log files were delayed until the actual request execution is completed. Recording or logging such messages were made using an in-memory model with delayed write, where buffers corresponding to such activities records the output as and when it happens and direct it only to the intended physical medium such as a file on disk at the end of a complete request execution cycle. This successfully prevented any delays being incurred by such activities on the request execution.

Secondly, priority inversions may result in an unexpected sequence of process execution which portrays a different view of the system activities than actually intended. For instance, a common debugging technique is the use of trace messages either on screen or written to a log file. When debugging the application, such trace messages will result in priority inversions where the sequence of events logged, will not be the actual sequence if not for the trace message itself. Therefore, such trivial debugging techniques cannot be used in the development phase of these systems. Instead, specialised tools such as the Thread Scheduling Visualiser [Oracle Corporation, 2009b] and memory based logging techniques that do not result in priority inversions have to be employed.

The memory based logging technique used, uses an in-memory buffer to store all logging information recorded at each modules along different points of execution of a request. At the end of an execution cycle, prior to a worker thread returning back to the thread pool, the logging information is written to a file. The system uses a single file per server and it is opened for writing at the initialisation of the logging module. This prevents any priority inversions at file creation. In our implementation the logging information was mostly data stored as strings. In the event of more complex data types needing to be stored and persisted, it is advised to use data structures with efficient constant time access to minimise the time spent on updating the log file with the information.

5.9 Summary

In this chapter, we presented a comprehensive discussion on how predictability of execution can be achieved in engineering web services middleware. We provided a set of guidelines that summarise the most important features such middleware must posses. These guidelines can be used to build web services middleware from ground-up or to enhance existing middleware to achieve predictability of execution. The guidelines mandate that web service requests be explicitly scheduled to meet a processing deadline and differentiation to be introduced by giving a higher priority to early deadlines. Moreover, they also highlight the importance of conditional acceptance of requests on the guarantee of meeting the processing deadline. Such specialised admission control and processing policies must be empowered by proper development libraries, development platforms and operating systems that could enforce required priority models in resource reservations and execution. Finally, the required changes and supporting activities around their integration into the middleware must ensure that such activities would not result in unexpected changes of priority, as in the case of priority inversion.

Several design principles and software engineering techniques that guarantees predictability of execution were presented and examples of how they were put to practice in existing middleware products were discussed as case studies. In applying these techniques, product specific implementation details were highlighted and generic steps pertaining to implementing such a feature were separated out for them to be directly applicable to any existing middleware product. Most web services middleware exhibit common core design features such as employing multiple thread pools for request execution.

Our approach for achieving predictability in such middleware was formed around such common features making this approach applicable to any web service middleware product.

The aforementioned techniques were firstly applied to an existing stand-alone web services middleware product. Next, the techniques were used in enhancing a cluster setup with the stand-alone middleware acting as the executors in the cluster. The functionality of the dispatcher was modified using the techniques described. Moreover, the change in its functionality considers predictability when dispatching decisions are made. Apart from the techniques that could be used in enhancing the middleware products, changes in supporting activities such as debugging is also required in ensuring predictability of execution. Few techniques that could be used for this purpose were also discussed in this chapter.

Chapter 6

Performance Modelling of EDF Scheduling in Web Services Middleware[‡]

This chapter presents a queueing theoretic performance model for a priority based multi-class preemptive $M/G/1$ system using EDF scheduling. A preemptive $M/G/1$ queueing model is the best representation for a stand-alone web services middleware using deadline based scheduling. Deriving a performance model allows an analytical study of its behaviour and facilitates optimisations without the need of an a real system. Additional performance attributes such as the mean waiting time of a request allows the comparison of EDF to other techniques that do not consider execution deadlines or laxity. Existing models of EDF scheduling systems consider it to be an $M/M/1$ queue or to be a non-preemptive $M/G/1$ queue. While, assuming the service times to be exponentially distributed results in simpler models, it is deemed unsuitable for web services workloads as a service could represent any type of processing. Supporting general service times allows a model to be valid for any type of workload. The model approximates the waiting time for a given priority class to be based on four parameters. Higher priority requests already present in the system, being executed prior to a request from the target class, lower priority requests already present in the system being executed prior to a request from the target class, higher priority requests arriving at

[‡] Preliminary versions of the work presented in this chapter will be published in [Gamini Abhaya et al., 2013].

the system after the target request and being serviced prior to it and the mean residual service time experienced by the priority class. Approximating additional time caused by preemptions that may happen in execution is a challenging task. This is achieved by estimating it as part of mean time for request completion for a given priority class and defining it as part of the mean time delay experienced due to jobs in execution, on an arrival. The model is evaluated for accuracy by obtaining analytical results and comparing them against results obtained by simulation. Results confirm that the model is indeed an accurate representation of the behaviour in such system with the difference between the results being a factor of 2 on average in high load conditions. Comparison of the model to other popular algorithms such as First-Come-First-Served, Round-Robin, Preemptive Priority Ordered and Non-Preemptive Priority Ordered reveal that EDF achieves a better balance of waiting times among priority classes where it favours high priority requests while preventing lower priority requests from over starvation. EDF achieves best waiting times for higher priority classes in lower to moderate loads (0.2 - 0.6) and records waiting times 6.5 times more than a static priority algorithm in high loads (0.9). However for the lowest priority classes it achieves comparable waiting times to Round-Robin and First-Come-First-Served in low to moderate loads and achieves waiting times only twice the amount of Round-Robin in high system loads.

6.1 Motivation

With the solutions presented in the previous chapters, algorithms and techniques that aid in achieving predictability of execution were discussed and evaluated using actual implementations. The attributes used for the evaluation were the percentage of requests accepted and the percentage of deadlines met by the systems. However, such attributes are not common in techniques that do not consider predictability as a performance goal. Instead, other common performance related attributes such as the waiting time experienced by a request, are considered.

This chapter presents a performance model for a EDF preemptive scheduling system that considers mean waiting time of a request as its primary performance attribute. Deriving such a performance model for a system has many benefits. Firstly, it is an analytical representation that can be used for studying the system behaviour, without the need of its physical presence. Secondly, it saves time and effort by allowing the evaluation of changes and optimisations to a system prior to its actual implementation. Finally, it

gives an overview of the behaviour of each task within the system, the effect of other requests on its execution and allows the estimation of the time it spends at each stage of the system, such as approximation of the mean waiting time experienced by a request from a particular priority class.

Existing work on performance models for EDF based systems has been based on queueing theory. [Kargahi and Movaghar, 2006] presents a non-preemptive $M/M/m/EDF$ and a preemptive $M/M/1/EDF$ model. The assumption they make on exponentially distributed service time may not be a suitable representation for web services workloads, as services could be used in exposing any type of system. [Chen and Decreusefond, 1996] present a non-preemptive $M/G/1/./EDF$ model which can be considered as a better representation of such workloads due to its validity for general workloads. The assumption of Poisson arrivals is an acceptable representation of the bursty nature of traffic on the Internet [Clark, 2000; Kresch and Kulkarni, 2011]. However, the system we thrive to model uses preemptive EDF scheduling.

6.2 Problem Statement

Deriving a queueing theory based performance model for a preemptive EDF scheduling system poses many challenges. The system considered must have stochastic properties. Therefore deterministic elements in the system such as the schedulability check cannot be represented in the model. Moreover, for a better representation of tasks and their deadlines, requests are considered to be grouped into priority classes. The priority of each class is determined by a static deadline offset assigned to every request of that class upon its arrival at the system. However, when priority order of requests are compared at runtime, the absolute deadlines of the requests are considered. Therefore, the system enforces the priorities dynamically and unlike static priority systems, a task from a priority class deemed to be lower priority may get execution preference over another due to its earlier deadline (despite the deadline offset being longer). Therefore, on a new task arrival, lower priority requests present in the system may receive service prior to the new arrival. The portion of lower priority requests that receive such service needs to be accounted for in the model. Similarly, not all higher priority requests arriving at the system after the target task may receive service prior to it. The portion of such higher priority requests needs to be estimated separately and accounted for in the performance model.

In a preemptive system, the execution of a task may be interrupted by the arrival of higher priority requests. Estimating the waiting time incurred by such preemptions is a complex task and the use of deadline based scheduling makes it even more challenging. Unlike with static priority based models, using EDF scheduling means that the estimates must consider preemptions based on dynamic priority enforcements at runtime, which means only a portion of the higher priority requests arriving while the target request is in execution maybe able to preempt it. This portion from each higher priority class needs to be estimated and accounted for in the model.

Addressing these important concerns, the research question attempted by this chapter is "How can a performance model be derived for a preemptive EDF scheduling system ?". In attempting a solution, the following main areas of concern are addressed.

- What are the dependant attributes on the execution of of a task to completion on a preemptive EDF based system?
- How can the completion time of a request be estimated considering preemptions, when using EDF scheduling where priorities are enforced dynamically?
- How can these different attributes be used together to derive a performance model for the overall system?

6.3 Outline of the Solution

The core contribution of this chapter is a performance model for an EDF based preemptive scheduling system with Poisson arrivals and a general service time distribution. [Chen and Decreusefond, 1996] is the only work of its kind that we found at the time of this research which considers an $M/G/1$ type system where EDF is used for non-preemptive scheduling of requests. The proposed model builds on their work to derive a system for a preemptive resume system (work-conserving). To the best of our knowledge, such a model has not been previously attempted at the time of this research. The significance of the model is supporting arbitrary service times, which makes it applicable to any type of workload as long as preemptive EDF scheduling is used in a work-conserving manner.

The model supports arbitrary number of request classes where their priorities are governed by the deadline offset assigned. A lower deadline offset signifies a higher priority.

The model is an approximation of the mean waiting time experienced by a request belonging to a particular priority class. The mean waiting time is defined based on four estimates. The time resulted by the execution of the portion of higher priority requests already found in the system by a newly arrived request belonging to the target priority class, and receive service prior to the target, the portion of lower priority requests already found in the system by the target request and receive service prior to the target, the portion of higher priority requests arriving at the system after the target request and receive service prior to the target and the mean time delay experienced by the target request as a result of jobs in execution at the time of arrival. The portions of requests for the first three parameters mentioned for every priority class are estimated based on the deadline difference between each of the priority classes and the target class.

The preemptions and their effect on the waiting time are estimated by defining it to be part of the mean delay incurred by the jobs in execution, at an arrival. This is done by introducing a definition for the mean completion time of a request which is defined as the sum of its mean service time and the time incurred by any preemptions. The additional time incurred by the preemptions are estimated using Little's law [Kleinrock, 1976] and two cases are considered based on how the deadline differences compare against the mean completion time. The mean completion time of the system is incorporated into the mean delay incurred by the executions at an arrival. It is estimated based on the probability of a request from a particular priority, be found in service by an arrival.

With this performance model, the goal is to achieve shorter waiting times for higher priority request classes and longer waiting times for lower priority classes. Given the general service times it supports, the proposed model is not only valid in the context of web services but in other systems where priority based preemptive EDF scheduling is used.

The rest of the chapter is organised as follows. First, a discussion on some of the related work in this area is presented. Next, Section 6.5 presents background information on the reference model the proposed model is based on. Thereafter, Section 6.6 presents the proposed model for a preemptive $M/G/1/./EDF$ system. It is followed by a theoretical proof of the model in Section 6.7. Following that in Section 6.8 comprehensive evaluation of the model is presented and compared with others. Section 6.9 provides a summary of the contribution.

6.4 Related Work

Previous attempts at using deadline based scheduling to differentiate service processing can be found in literature. In [Dag and Gokgol, 2006] EDF is used for scheduling packets for transmission where the deadline is used as the QoS parameter. The transmission buffer gets sorted according to the deadline and empirical results reveal that EDF minimises loss rates of packets due to deadline violations. [Li et al., 2007] proposes a non-preemptive EDF based algorithm that groups tasks with deadlines closer together and use Shortest Job First scheduling within the group, while using EDF among the different groups. By grouping tasks with similar deadlines, they try to minimize the loss rate under various workloads. Similar to the proposed solutions discussed in the previous chapters, both these works considers the deadline loss rate as the performance attribute.

[Kargahi and Movaghar, 2006] presents a method for performance analysis of EDF scheduling with deadlines at the beginning of service and deadlines at the end of service. They consider a non-preemptive $M/M/m/EDF + G$ system with deadlines at the beginning of service and a preemptive $M/M/1/EDF + G$ system for their analysis. They make the assumption that service times are exponentially distributed, arrivals happen according to a Poisson process and consider generally distributed deadlines in each system. The system is modelled as a Markov chain and the loss rate is considered to be the main quality of service measure. The optimality of EDF and the known results for FCFS are considered as the upper and lower bound respectively for the loss rate. These two are combined using a multiplier and the fraction of jobs missed in each case is obtained through simulation and analytical results.

Work of [Lehoczky, 1996] tries to incorporate the laxity property of a task into a queueing model. The system is modelled as an $M/M/1$ system and the loss rate is considered as the main QoS attribute. The state of the system is represented by the laxities or lead times of the tasks in the system. The devised model is a Markov process on the transitions of lead times in the system upon arrivals and completions of tasks. The model has been evaluated in heavy traffic conditions by the loss rate the system results in. The evaluations confirms the importance the scheduling or the queueing discipline in meeting the task deadlines. The main aim of this work has been to bridge the gap between real-time systems and queueing theory where the former is often associated with task sets that are more deterministic, while the latter deals with stochastic systems. This

work is further extended to $GI/M/1$ systems in [Lehoczky, 1997]. Their assumption of exponentially distributed service times may not be a proper representation of some of the web service workloads. Web services can have arbitrary service times as they could be used to expose any type of system.

The work of [Chen and Decreusefond, 1996] considers a multi-class $M/G/1/$ system where EDF is used to schedule requests among an arbitrary number of classes with soft deadlines. They consider the system to be non-preemptive and base their model on the non-preemptive $M/G/1$ definition found in [Kleinrock, 1976]. They derive an analytical model based on it to approximate the waiting time of such a system. The model results in a series of equations that approximate the waiting time for each priority class. It is proven to be valid using an iterative process and further proof is provided by comparing analytical results with results obtained from simulation. This particular model is the only of its kind that we found to be using EDF based scheduling with a $M/G/1$ queue. Its significance is the fact that the model does not depend on the service time distribution of the requests. Therefore, it is valid for any type of traffic when scheduled with EDF within conditions mentioned. Given the model is valid for only a non-preemptive system, it is not a complete representation of the system we thrive to model. However, the proposed solution is built on the basics of their model and extended for a preemptive scheduling system.

6.5 Background

We consider a preemptive-resume $M/G/1$ queueing system that is work conserving, which receives multiple Poisson traffic streams. This system shares some common characteristics with a non-preemptive version. [Kleinrock, 1976] contains a generic framework, which helps in calculating mean waiting times for a non-preemptive, work conserving $M/G/1$ systems, with multiple request streams. This framework is introduced in this section and used subsequently, in our analytical model.

We consider a system of multiple (N number of) Poisson request streams and study the system in the point of view of a newly arrived task. We refer to this newly arrived task as the *tagged* task. Let us consider the tagged task to be from request stream i where $i = (1, 2, 3..N)$. The request streams have a priority ordering, such that i has a higher priority than j if $i < j$. Upon its arrival at the system, the tagged task may find a task already in execution and tasks that have arrived before, queued, waiting for its turn for

execution. Due to the enforced priorities, some of the existing tasks may be executed prior to the tagged. Similarly, tasks that arrive at the system after the tagged task, may also be executed prior to it. Considering these the following notations are used for the system,

- $\overline{W_0}$ represents the mean residual service time, or the mean time required to complete the execution of the task currently in service at the time of the arrival of the tagged task from stream i.
- $\overline{N_{j,i}}$ represents the mean number of tasks from stream j which arrived at the system *prior* to our tagged task from stream i and starts execution *prior* to our tagged task.
- $\overline{M_{j,i}}$ represents the mean number of tasks from stream j which arrive at the system while our tagged task from stream i is waiting in the queue (i.e. *after* our tagged task), but starts execution *prior* to the tagged task.
- λ_i represents the arrival rate for stream i.
- ρ represents the total load experienced by the system.
- $E(x_i)$ represents the mean of the service time distribution.
- $E(x_i^2)$ represents the second moment of the service time distribution.

For a $M/G/1$ system, $\overline{W_0}$ is defined by the following formula,

$$\begin{aligned} \overline{W_0} &= \sum_{i=1}^{N} \rho \frac{E(x_i^2)}{2E(x_i)} \\ &= \sum_{i=1}^{N} \frac{\lambda_i E(x_i^2)}{2} \end{aligned} \tag{6.5.1}$$

Consequently, the mean waiting time for the tagged task from stream i can be defined as,

$$\overline{W_i} = \overline{W_0} + \sum_{j=1}^{N} E(x_j)(\overline{N_{j,i}} + \overline{M_{j,i}}) \tag{6.5.2}$$

This method of calculating the mean waiting time is valid across all priority queueing disciplines that support general service times. The solution procedure contains two steps, in which firstly the estimation of $N_{j,i}$ and $M_{j,i}$ will depend on the service disciplines. Subsequently, the solution can be achieved with a resulting set of equations from (6.5.2).

6.6 The Proposed Model

This section presents the proposed model to evaluate the performance of a multi-class $M/G/1$ deadline based scheduling system. The evaluation focuses on mean waiting time of multiple streams of requests being serviced by a single server. As the proposed model is based on the work by [Chen and Decreusefond, 1996], the same system configuration is followed. Each request stream has its own deadline and the requests are serviced and queued in the order of their increasing deadlines. We use the terms job and task synonymously to refer to a request that needs to be serviced and the term class and stream synonymously to refer to a request stream.

The system has N number of independent streams through which requests arrive at the system following a Poisson process, where each stream is identified by $i, i = 1, 2, ...N$. Each stream is associated with a different deadline and all requests from the same stream gets assigned a constant deadline offset. For instance, a request from stream i arriving at the system at time t will have a deadline offset of $t + d_i$ where d_i is a constant for stream i. The priority of a stream is decided by the associated deadline offset where stream j, if $i < j \Rightarrow d_i \leq d_j$. Moreover the difference between deadline offsets of two streams are denoted by $D_{j,i} = d_j - d_i$. Scheduling of jobs among different input streams uses EDF and jobs originating from the same stream are treated in a First-Come-First-Served basis.

The following symbols and notations are used in the text in describing the proposed model.

Symbol	Description
λ_i	Arrival rate of stream i jobs (Poisson arrivals).
$\overline{X_i}$	Mean of the service time of the distribution for class i.
$\overline{X_i^2}$	Second moment of the service time distribution.
ρ_i	Utilisation of the server by jobs belonging to stream / priority i. $[\rho_i = \lambda_i \overline{X_i}]$
$\overline{W_i}$	Mean waiting time for a request of stream / priority i.
$\overline{C_i}$	Mean time required to complete service, for a request of stream / priority i including preemptions, i.e. Time between starting service and finishing service including preemptions.
$\overline{W_0}^i$	Mean time delay experienced by an arrival from stream i, from the jobs already in progress.
$\overline{R_i}$	Mean residual service time for a request of stream / priority i. $[\overline{R_i} = \frac{X_i^2}{2X_i}]$ for a $M/G/1$ system.
$\overline{N_{j,i}}$	Mean number of jobs from stream j which arrive *before* tagged request from stream i and receive service *prior* to the tagged request.
$\overline{M_{j,i}}$	Mean number of jobs from stream j which arrive *after* tagged request from stream i and receive service *prior* to the tagged request.
$D_{j,i}$	Difference in the deadline offsets of streams i and j. $[D_{j,i} = d_j - d_i]$.

Table 6.1: Symbols and Notations used in the proposed model

The mean waiting time experienced by a given priority class is considered to be the main performance attribute approximated by the proposed model. Only waiting time is considered in this model as it is the most commonly used metric to describe the performance of a system. As such, the performance of EDF could be compared with a multitude of algorithms and techniques in used in similar systems. The aim of the model is to approximate the waiting time of any given priority class and define it, based on the impact of other tasks either queued for service or executing in the system.

We derive the proposed model in two parts. Firstly, we follow the method presented in [Chen and Decreusefond, 1996] to derive an initial overall equation for the system that identifies the parameters based on the generic equation given by 6.5.2. Secondly, we approximate the time incurred by preemptions as part of the definition for mean time delay experienced by a priority class, due to the jobs currently in execution and it is presented separately. Since EDF is used for scheduling requests, the decision made by the server in selecting the next request to be executed is independent of its service time. Therefore, to arrive at the initial overall equation that estimates waiting time of a request stream, we follow the equation 6.5.2. We estimate the parameters $N_{j,i}$ and $M_{j,i}$

based on absolute deadlines considered by EDF scheduling.

6.6.1 Deriving equations for case $N = 2$

We consider the simplest scenario of having two request streams where $N = 2$ and observe the following on its behaviour. We consider stream 1 to have a higher priority than stream 2 as requests from stream 1 have shorter deadlines than requests from stream 2 $t + d_1 < t + d_2$. In a typical priority based system that uses Head-of-line queueing discipline [Kleinrock, 1976] higher priority tasks are always serviced ahead of lower priority tasks. Given that the priority is determined by the absolute deadline in the system under consideration, EDF scheme allows the processing of a portion of requests from stream 2 ahead of a newly arrived request from stream 1. Requests belonging to the same stream albeit being deadline ordered exhibits a service order of FCFS due to the absolute deadlines being considered (due to the same deadline offset used for all stream members).

Firstly, we consider the viewpoint of a tagged request belonging to stream 1:

- All requests belonging to stream 1 found in the queue by the tagged request upon its arrival at the system, will be serviced prior to the tagged request. Requests from stream 1 arriving at the system after the tagged request will be served later. Therefore, $N_{1,1} = \lambda_1 \overline{W_1}$ and $M_{1,1} = 0$.
- Requests from stream 2 that are already present in the queue may have absolute deadlines that are prior to the absolute deadline of the tagged request. While there will be on average $\lambda_2 \overline{W_2}$ requests from stream 2 already present in the queue, only requests arriving *at least* $D_{2,1}$ time before and still in the queue after $D_{2,1}$ time, will receive service prior to the tagged request. Figure 6.1 illustrates this scenario. Consider *I* to be the tagged request from stream 1. Out of all stream 2 requests already in the queue, request *J* is the one with the latest absolute deadline that gets service prior to the tagged request. Therefore, a request from stream 2 to be serviced prior to tagged must have arrived at the system at least $D_{2,1}$ time units before the tagged request and have waited in the queue $D_{2,1}$ amount of time. Considering this, $N_{2,1} = max(0, \lambda_2(\overline{W_2} - D_{2,1}))$ can be estimated.
- Due to the difference in the deadline offsets $d_1 < d_2$ requests from stream 2 arriving at the system after the tagged request will be executed after the tagged.

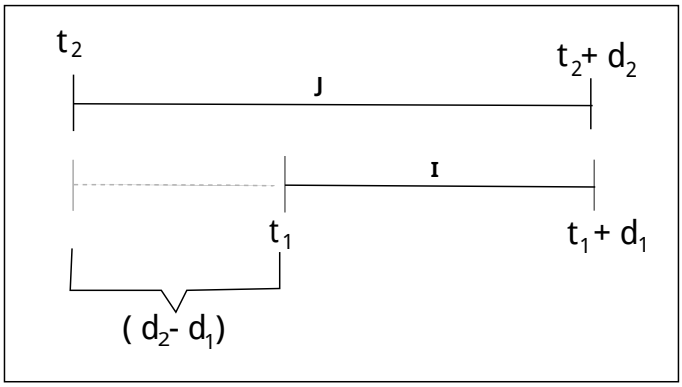

Figure 6.1: Deadline difference between requests

Hence, $M_{2,1} = 0$.

Next we consider the viewpoint of the tagged request from stream 2:

- Due to the difference in the deadline offsets $d_1 < d_2$ stream 1 requests already present in the queue when the tagged request arrives at the system, will be serviced prior to the tagged. Therefore, $N_{1,2} = \lambda_1 \overline{W_1}$.
- Once the tagged request arrives at the system, a portion of the stream 1 requests arriving subsequently at the system will have deadlines earlier than the tagged request. With reference to Figure 6.1, if the request J is the tagged request, I can be considered as the last stream 1 request that arrive thereafter and receive service prior to the tagged. It is such that any stream 1 request that arrive at the system after the tagged request *no later* than $D_{2,1}$, will receive service prior to the tagged. However, given the waiting time of stream 2, the tagged request maybe in the queue for a time period less than $D_{2,1}$, given that $\overline{W_2} < D_{2,1}$. Considering these, it can be estimated that $M_{1,2} = \lambda_1 min(\overline{W_2}, D_{2,1})$.
- Due to the constant deadline offset of a request stream, all of the already queued requests from stream 2 will be executed prior to the tagged requests. Similarly, requests from the same stream arriving at the system after the tagged request will have an absolute deadline that is later than the tagged. Hence, all such requests will only be serviced after the tagged request. Therefore, it can be easily concluded that $N_{2,2} = \lambda_2 \overline{W_2}$ and $M_{2,2} = 0$.

Next, we substitute these estimates directly in Equation 6.5.2 and receive the following

two equations that estimates the waiting times for streams 1 and 2:

$$\begin{aligned}
\overline{W_1} &= \overline{W_0} + \overline{X_1}\lambda_1\overline{W_1} + \overline{X_1}\; max(0, \lambda_2(\overline{W_2} - D_{2,1})) \\
&= \overline{W_0} + \rho_1\overline{W_1} + \rho_2\; max(0, (\overline{W_2} - D_{2,1})) \\
\overline{W_2} &= \overline{W_0} + \overline{X_1}\lambda_1\overline{W_1} + \overline{X_2}\lambda_2\overline{W_2} + \overline{X_1}\lambda_1\; min(W_2, D_{2,1}) \\
&= \overline{W_0} + \rho_1\overline{W_1} + \rho_2\overline{W_2} + \rho_1\; min(\overline{W_2}, D_{2,1})
\end{aligned}$$

6.6.2 Deriving the generic equation

Parts of the two equations we achieved above can be generalised for the case of more than two streams. The reasoning provided for obtaining the two equations stands the same for any number of levels or equations. The equation for the waiting time of each request stream is made up of four distinct parts. The mean residual service time represented by $\overline{W_0}$ is common to all request streams. The remaining three parts can be generalised for the rest of request streams, from the view point of a tagged request from stream i.

From the earlier discussion we can conclude that all requests from higher priority streams that arrive prior to the tagged request will receive service before the tagged request. We could generalise the number of such requests as follows,

$$N_{k,i} = \lambda_k\overline{W_k} \qquad\qquad 1 \leq k \leq i$$

A portion of requests from lower priority streams that arrive at the system prior to the tagged request, will receive service prior to the tagged due to the earlier deadlines they posses. The number of such requests can be estimated for any request stream with the following,

$$N_{k,i} = \lambda_k\; max(0, \overline{W_k} - D_{k,i}) \qquad\qquad i < k \leq N$$

The remaining part of the equation is the representation of requests from higher priority streams that arrive at the system after the tagged request and receive service before the tagged request, due to the earlier deadlines they posses. This number could be estimated

with the following,

$$M_{k,i} = \lambda_k \, min(\overline{W_i}, D_{i,k}) \qquad\qquad 1 \leq k < i$$

In turn these generic terms could be used in Equation 6.5.2 to arrive at the generic equation for the waiting time of any stream i.

$$\overline{W_i} = \overline{W_0}^i + \sum_{k=1}^{i} \rho_k \overline{W_k} + \sum_{k=i+1}^{N} \rho_k max(0, \overline{W_k} - D_{k,i}) + \sum_{k=1}^{i-1} \rho_k min(\overline{W_i}, D_{i,k}) \quad (6.6.1)$$

The term $\overline{W_0}$ has been replaced by $\overline{W_0}^i$ as the system under consideration is a preemptive resume system. $\overline{W_0}^i$ is the mean residual service time of the system for the i th priority. Herein, only residual service times of streams 1 to i are considered as only a higher priority request (k where $1 \leq k < i$) may preempt a request (of stream i) in execution.

6.6.3 Estimation of the mean delay incurred by jobs in execution

In Equation 6.6.1, $\overline{W_0}^i$ represents the mean delay incurred by the jobs in execution, at an arrival. While this estimation is straightforward [Kleinrock, 1976] in a non-preemptive system, a preemptive scheduling system tends to be more complex. Herein, we consider the system to be preemptive resume where a request of lower priority in execution may be preempted by a newly arrived request with a higher priority and resumes execution later. With EDF scheduling, the priority of a request is decided at runtime by its absolute deadline, such that the request with the highest priority is the one with the earliest deadline, at any given time.

For such a $M/G/1/./EDF$ system we derive the mean residual service time as follows. We define the random variable $\overline{C_i}$ as the mean time required to complete service for a request from stream i, including the time the request is preempted. During an average service completion interval of a stream i request, the mean number of stream j jobs (where stream j is of higher priority and can preempt a class i job in execution) that

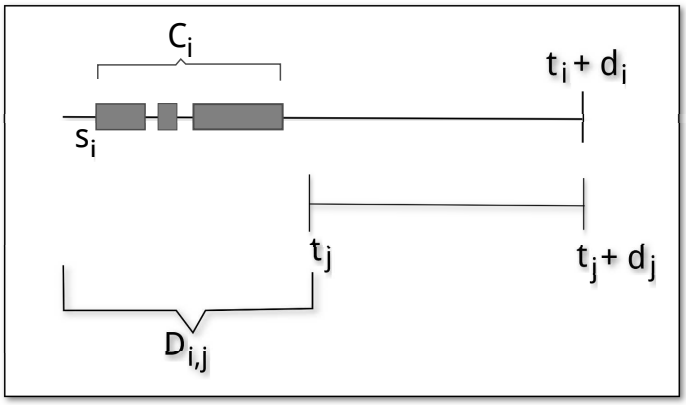

Figure 6.2: Request completing execution within $D_{i,j}$

arrive at the system can be estimated as $\lambda_j \overline{C_i}$. If all such class j requests preempt the class i request in service and execute until completion, $\overline{C_i}$ could be defined as,

$$\overline{C_i} = \overline{X_i} + \sum_{j=1}^{i-1} (\lambda_j \overline{C_i X_j}) \tag{6.6.2}$$

The execution of a request may happen in a staggered manner due to preemptions that take place. Recall that the use of EDF scheduling enforces the priorities dynamically at runtime based on absolute deadlines. As such, only a portion of class j requests are able to preempt an already executing class i request. Therefore, a request from stream j will be faced with the following two conditions for a preemption to happen. As illustrated in Figure 6.2, it is possible for a request to complete execution with preemptions within the time period $D_{i,j} = (d_i - d_j)$. In this instance, tasks with earlier deadlines (compared to the stream i task in execution) that arrive within the time period $(D_{i,j} - \overline{C_i})$ will not result in the preemption of request i, as it has already completed execution.

Figure 6.3 illustrates a scenario where the staggered execution of the request from stream i continuing beyond the time period $D_{i,j}$. In this instance, tasks from stream j arriving within the period $\overline{C_i}$, but beyond $D_{i,j}$ would not result in the preemption, as their absolute deadlines would be later than that of the task in execution at the time. Therefore, we could assume that the preemption of a request in execution from stream

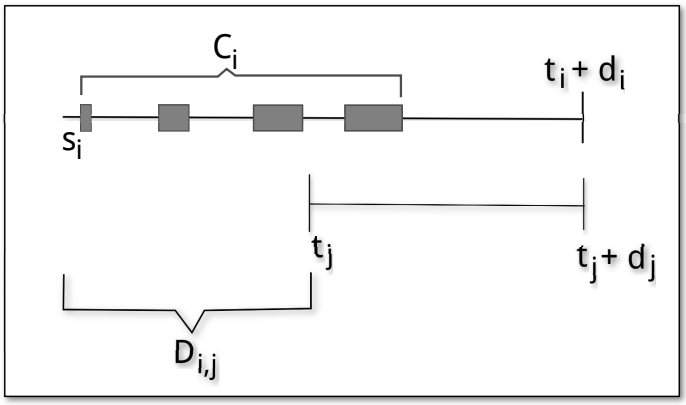

Figure 6.3: Request completing execution beyond $D_{i,j}$

i would only take place within the least of $\overline{C_i}$ and $D_{i,j}$ for any given i and j.

Considering the mentioned scenario, Equation 6.6.2 for the completion time for a class i request, could be modified as follows:

$$\overline{C_i} = \overline{X_i} + \sum_{j=1}^{i-1} (\lambda_j \; min(D_{i,j}, \overline{C_i}) \; \overline{X_j})$$
$$\overline{C_i} = \overline{X_i} + \sum_{j=1}^{i-1} (\rho_j \; min(D_{i,j}, \overline{C_i})) \tag{6.6.3}$$

Let j^* = subscript $j = 1, 2, ..i-1$ such that $D_{i,j} \leq \overline{C_i}$
Let $j^{'}$ = subscript $j = 1, 2, ..i-1$ such that $D_{i,j} > \overline{C_i}$

$$\begin{aligned}
\overline{C_i} &= \overline{X_i} + \sum_{j^*}(\rho_{j^*} D_{i,j^*}) + \sum_{j'}(\rho_{j'}\overline{C_i}) \\
\overline{C_i}(1 - \sum_{j'}\rho_{j'}) &= \overline{X_i} + \sum_{j^*}(\rho_{j^*} D_{i,j^*}) \\
\overline{C_i} &= \frac{\overline{X_i} + \sum_{j^*}(\rho_{j^*} D_{i,j^*})}{(1 - \sum_{j'}\rho_{j'})}
\end{aligned} \tag{6.6.4}$$

Let P_i be the probably of a request from stream i being in service at an arrival. P_i could be defined as:

$$P_i = \lambda_i \overline{C_i} \tag{6.6.5}$$

Substituting 6.6.4 in 6.6.5,

$$\begin{aligned}
P_i &= \lambda_i \left(\frac{\overline{X_i} + \sum_{j^*}(\rho_{j'} D_{i,j^*})}{(1 - \sum_{j'}\rho_{j'})} \right) \\
&= \frac{\rho_i + \sum_{j^*}(\rho_{j'} D_{i,j^*})\lambda_i}{(1 - \sum_{j'}\rho_{j'})}.
\end{aligned} \tag{6.6.6}$$

The mean delay experienced by a new arrival from the jobs already in execution, can be defined as the sum of all probabilities of a job of a given class is in service, times the mean residual service time for the given class [Kleinrock, 1976]. Therefore, we could define $\overline{W_0}^i$,

$$\overline{W_0}^i = \sum_{k=1}^{i} \left(P_k \overline{R_k} \right) \tag{6.6.7}$$

Substituting 6.6.6 in 6.6.7,

$$\overline{W_0}^i = \sum_{k=1}^{i} \overline{R_k} \left(\frac{\rho_i + \sum_{j^*}(\rho_{j'} D_{i,j^*})\lambda_i}{(1 - \sum_{j'}\rho_{j'})} \right) \tag{6.6.8}$$

where $\overline{R_i} = \frac{\overline{X_i}^2}{2\overline{X_i}}$ for an $M/G/1$ system [Kleinrock, 1976]. Note that when the two

extreme cases are considered,

If $\overline{C_i} \leq D_{i,j}\ \forall j$, then $\overline{W_0}^i = \sum_{k=1}^{i} \overline{R_i} \left(\frac{\rho_i}{(1-\sum_j \rho_j)} \right)$

and if $\overline{C_i} > D_{i,j}\ \forall j$, then $\overline{W_0}^i = \sum_{k=1}^{i} \overline{R_i} \left(\rho_i + \sum_j (\rho_j D_{i,j}) \lambda_i \right)$

In conclusion, the following generic expression estimates the mean waiting time for a given class i in a $M/G/1/./EDF$ system.

$$\overline{W_i} = \overline{W_0}^i + \sum_{k=1}^{i} \rho_k \overline{W_k} + \sum_{k=i+1}^{N} \rho_k max(0, \overline{W_k} - D_{k,i}) + \sum_{k=1}^{i-1} \rho_k min(\overline{W_i}, D_{i,k}) \quad (6.6.9)$$

Given the preemptive resume scheduling discipline considered, the mean waiting times must satisfy the conservation law for preemptive resume $M/G/1$ queues [Bolch et al., 2006; Kleinrock, 1976].

$$\sum_{k=1}^{i} \rho_k \overline{W_k} = \frac{\sigma_i \overline{W_0}^i}{1 - \sigma_i} \quad (6.6.10)$$

where $\sigma_i = \sum_{k=1}^{i} \rho_k$. The waiting time for each priority level could be estimated by using equation 6.6.9 and solving the resultant set of non-linear equations under the constraint given by equation 6.6.10.

6.7 Theoretical Analysis

This section presents a theoretical analysis of the proposed model presented in the earlier section. We prove that the system of N non-linear equations given by 6.6.9 has exactly one solution, following the solution strategy used by [Chen and Decreusefond, 1996]. First, we transform equation 6.6.9 in to the following form using the conservation law specified in equation 6.6.10.

$$\begin{aligned}\overline{W_i} &= \overline{W_0}^i + \sum_{k=1}^{i} \rho_k \overline{W_k} + \sum_{k=i+1}^{N} \rho_k \, max(0, \overline{W_k} - D_{k,i}) + \sum_{k=1}^{i-1} \rho_k \, min(\overline{W_i}, D_{i,k}) \\ &= \left[\overline{W_0}^i + \frac{\sigma_i \overline{W_0}^i}{1-\sigma_i} + \sum_{k=i+1}^{N} \rho_k \, max(0, \overline{W_k} - D_{k,i}) \right] + \sum_{k=1}^{i-1} \rho_k \, min(\overline{W_i}, D_{i,k}) \\ &= \left[\frac{\overline{W_0}^i}{1-\sigma_i} + \sum_{k=i+1}^{N} \rho_k \, max(0, \overline{W_k} - D_{k,i}) \right] + \sum_{k=1}^{i-1} \rho_k \, min(\overline{W_i}, D_{i,k})\end{aligned} \tag{6.7.1}$$

It can be observed that the waiting time calculation using the above equation requires an iterative process. We consider a particular class $I, 1 < I \leq N$ and make the assumption that $\overline{W_k}, I < k \leq N$ are known. With this assumption, all the terms bracketed in equation 6.7.1 are known and can be considered constant. Mean waiting time $\overline{W_I}$ can be estimated by solving the equation in the form of,

$$X = C + \sum_{1 \leq j < I} a_j \, min(X, b_j) \tag{6.7.2}$$

where a_j, b_j and C are known to be positive constants. Moreover, it is also known that $\sum_{j=1}^{I-1} a_j < 1$ (as $\sum_{j=1}^{I-1} \rho_j < \rho < 1$ is known). By transforming the equation to the above form, which is the starting point for the proof presented in [Chen and Decreusefond, 1996], it can be concluded that the same proof holds equally true for equation 6.7.1 and it has only a single solution. Similarly, the most direct way of solving the set of equations is also to following the iterative process as outlined in [Chen and Decreusefond, 1996].

6.8 Evaluation

We evaluate the proposed model for accuracy of estimation, using analytical and simulation results. The analytical evaluation is done by manually calculating the waiting times using equation 6.6.9.

6.8.1 Theoretical Evaluation

The model is evaluated for the following scenario. The computations demonstrate the iterative process involved with solving the set of equations. The system considered for the evaluation has three priority levels.

Deadlines	
d_1	1500 ms
d_2	4000 ms
d_3	6000 ms
Deadline differences	
$D_{2,1}$	2500 ms
$D_{3,1}$	4500 ms
$D_{3,2}$	2000 ms
Arrival Rates	
λ_1	0.0005 ms^{-1} (5 tasks per second)
λ_2	0.0002 ms^{-1} (2 tasks per second)
λ_3	0.0001 ms^{-1} (1 tasks per second)

Service Times	
$\overline{X_1}$	502.5 ms
$\overline{X_2}$	1502.5 ms
$\overline{X_3}$	2502.5 ms
System Load	
ρ_1	0.25125
ρ_2	0.3005
ρ_3	0.25025
Second Moments	
$\overline{X_1^2}$	335673
$\overline{X_2^2}$	2340673
$\overline{X_3^2}$	6345673

Table 6.2: Sample parameters

We initiate the process by calculating the components needed to find the mean delay incurred by tasks in execution, at an arrival ($\overline{W_0}^i$). Using equation 6.6.4,

For $i = 1$,
$\overline{C_1} = \overline{X_1} = \underline{502.5}$

For $i = 2$,
$\overline{C_2} = \overline{X_2} + \rho_1 \; min(2500, \overline{C_2})$

The iterative process to calculate C_2 is started by considering it to be 0 on the right side of the equation. The resultant value for C_2 is used thereafter for the right side of the equation in the subsequent iterations until the results converge on a single value.

1st iter.	$\overline{C_2}$	= 1502.5	
2nd iter.	$\overline{C_2}$	= $1502.5 + 0.25125 \times 1502.5$	= 1880
3rd iter.	$\overline{C_2}$	= $1502.5 + 0.25125 \times 1880$	= 1974.85
...			
9th iter.	$\overline{C_2}$	= $1502.5 + 0.25125 \times 2006.64$	= 2006.66
10th iter.	$\overline{C_2}$	= $1502.5 + 0.25125 \times 2006.66$	= $\underline{2006.67}$

Similarly for $i = 3$,
$\overline{C_3} = \overline{X_3} + \rho_1\ min(4500, \overline{C_3})$ + $\rho_2\ min(2000, \overline{C_3})$

1st iter.	$\overline{C_3}$	= 2502.5	
2nd iter.	$\overline{C_3}$	= $2502.5 + 0.25125 \times 2502.5 + 0.3005 \times 2000$	= 3732.25
3rd iter.	$\overline{C_3}$	= $2502.5 + 0.25125 \times 3732.25 + 0.3005 \times 2000$	= 4041.23
...			
9th iter.	$\overline{C_3}$	= $2502.5 + 0.25125 \times 4144.8 + 0.3005 \times 2000$	= 4144.88
10th iter.	$\overline{C_3}$	= $2502.5 + 0.25125 \times 4144.88 + 0.3005 \times 2000$	= $\underline{4144.9}$

Next, we substitute these values in equation 6.6.5,

P_1 = 0.0005×502.5
= $\underline{0.25125}$
P_2 = 0.0002×2006.67
= $\underline{0.40133}$
P_3 = 0.0001×4144.9
= $\underline{0.41449}$

With these values, we can calculate the mean delay incurred by tasks in execution, for each stream using equation 6.6.7,

$$\begin{aligned}
\overline{W_0}^1 &= P_1\overline{R_1} \\
&= 0.25125 \times \tfrac{335673}{2\times502.5} \\
&= \underline{83.918} \\
\overline{W_0}^2 &= P_1\overline{R_1} + P_2\overline{R_2} \\
&= 0.25125 \times \tfrac{335673}{2\times502.5} + 0.40133 \times \tfrac{2340673}{2\times1502.5} \\
&= \underline{396.518} \\
\overline{W_0}^3 &= P_1\overline{R_1} + P_2\overline{R_2} + P_3\overline{R_3} \\
&= 0.25125 \times \tfrac{335673}{2\times502.5} + 0.40133 \times \tfrac{2340673}{2\times1502.5} + 0.41449 \times \tfrac{6345673}{2\times2502.5} \\
&= \underline{922.036}
\end{aligned}$$

Next we calculate the following,

$$\begin{aligned}
\tfrac{\overline{W_0}^1}{1-\sigma_1} &= \tfrac{83.918}{0.74875} \\
&= \underline{112.08} \\
\tfrac{\overline{W_0}^2}{1-\sigma_2} &= \tfrac{396.518}{0.44825} \\
&= \underline{884.59} \\
\tfrac{\overline{W_0}^3}{1-\sigma_3} &= \tfrac{922.036}{0.198} \\
&= \underline{4656.747}
\end{aligned}$$

These values can be directly used for the final step of calculating the individual waiting time. We choose equation 6.7.1 in place of equation 6.6.9, as it contains more constants. Herein, the calculation is an iterative process which must be done in the reverse order of priorities i.e. starting from the priority 3, then backwards.

For $i = 3$,

$\overline{W_3} = 4656.747 + \rho_1\ min(\overline{W_3}, 4500) + \rho_2\ min(\overline{W_3}, 2000)$

As done in the iterative process earlier, we start the process with 0 for the value of W_3 on the right side of the equation.

1st iter.	$\overline{W_3}$	$= 4656.747$	
2nd iter.	$\overline{W_3}$	$= 4656.747 + 0.25125 \times 4500 + 0.3005 \times 2000$	$= 6388.372$
3rd iter.	$\overline{W_3}$	$= 4656.747 + 0.25125 \times 4500 + 0.3005 \times 2000$	$= \underline{6388.372}$

With the estimation for $\overline{W_3}$, we proceed to calculate $\overline{W_2}$,

For $i = 2$,

$\overline{W_2} = 884.59 + \rho_3\ max(0, 6388.372 - 2000) + \rho_1\ min(\overline{W_2}, 2500)$

1st iter.	$\overline{W_2}$	$= 884.59 + 0.25025 \times 4388.372$	$= 1982.78$
2nd iter.	$\overline{W_2}$	$= 884.59 + 0.25025 \times 4388.372 + 0.25125 \times 1982.78$	$= 2480.95$
3rd iter.	$\overline{W_2}$	$= 884.59 + 0.25025 \times 4388.372 + 0.25125 \times 2480.95$	$= 2606.12$
4th iter.	$\overline{W_2}$	$= 884.59 + 0.25025 \times 4388.372 + 0.25125 \times 2500$	$= 2610.91$
5th iter.	$\overline{W_2}$	$= 884.59 + 0.25025 \times 4388.372 + 0.25125 \times 2500$	$= \underline{2610.91}$

All terms known, $\overline{W_1}$ could be calculated,

For $i = 1$,
$\overline{W_1} = 112.08 + \rho_2\ max(0, 2610.91 - 2500) + \rho_3\ max(0, 6388.372 - 4500)$

$$\begin{aligned}\overline{W_1} &= 112.08 + 0.3005 \times 110.91 + 0.25025 \times 1888.372 \\ &= 112.08 + 33.33 + 472.565 \\ &= \underline{617.975}\end{aligned}$$

Results

$\overline{W_1}$ = 617.98 ms

$\overline{W_2}$ = 2610.91 ms

$\overline{W_3}$ = 6388.37 ms

With the results obtained, it can be observed that priority class 1 which has the highest priority, receives the best service with the lowest waiting time among the classes. As intended, priority classes 2 and 3 have the higher waiting times with class 3 having the longest. This is due to requests from class 2 and 3 in execution being preempted by higher priority classes arriving at the system due to their earlier deadlines.

6.8.2 Empirical Evaluation

The evaluation of the proposed model was carried out using direct substitution (analytical) and comparing it with a simulated version of the scenario. The waiting times obtained in the analytical evaluation, by value substitution in the proposed model were compared against waiting times measured by simulations using Omnett++ [Varga, 2001, 2010], a discrete event simulator.

The main metric measured and used for comparisons between the two methods is the mean waiting time for each priority class and for the overall system. As a secondary parameter, we also measured the deadline miss rate reported for each priority class. We

consider systems with 2 priority classes up to a maximum of 5 priorities for brevity. Each system was tested for five load conditions. We considered the system load $\rho < 1$ to be 0.3 for lowest and 0.9 as the highest. The total load is divided amongst the priority classes in the reverse order so that the lowest numbered priority class is responsible for the highest load and in turn has the highest priority in the system. For instance, a 3 priority level system with 0.6 as the system load will have load distributed in a 3:2:1 ratio $\rho_1 = 0.3, \rho_2 = 0.2$ and $\rho_3 = 0.1$. As our estimate is independent of the service distribution, we use uniformly distributed service times for most cases. Furthermore, we also use exponentially distributed service times for some of the simulations to analyse the estimates for different service time distributions.

Arrival rates are calculated based on the load and mean service times $\left(\lambda_i = \frac{\rho_i}{\overline{X_i}}\right)$. Deadlines were picked considering the maximum service time values possible for each priority class. Performance measures on each simulation were a data set of 50,000 requests. Simulations involved a warm-up period of 10,000 to let the system arrive at a steady state prior to collection of data and a cool-down period of 5000 requests to ensure the measurements were not influenced by such phenomenon [Heidelberger, 1995].

6.8.3 Analytical Results

As the system is modelled as a priority based preemptive resume $M/G/1/./EDF$ system, the following characteristics were expected from the waiting time estimates obtained from the model. The system represented by the proposed model favours shorter deadlines and considers them to be higher priority enabling them to have shorter waiting times over other requests in the system. However, with deadlines being the deciding factor ensures that lower priority classes does not starve, as in the case of traditional static priority based preemptive systems.

Figures 6.4 contains plots of the estimated waiting times for 4 systems with the number of priority classes ranging from 2 to 5. Tables 6.3, 6.4 and 6.5 contains the detailed waiting times estimated. The estimates for each class follows the intended behaviour of shorter waiting times for higher priority and longer waiting times for lower priority classes. Moreover, the difference in waiting times between priorities gets larger with the total load in the system. At higher loads, lower priority requests have to wait longer for processing resources. The difference in arrival rates between the classes is the cause for this behaviour.

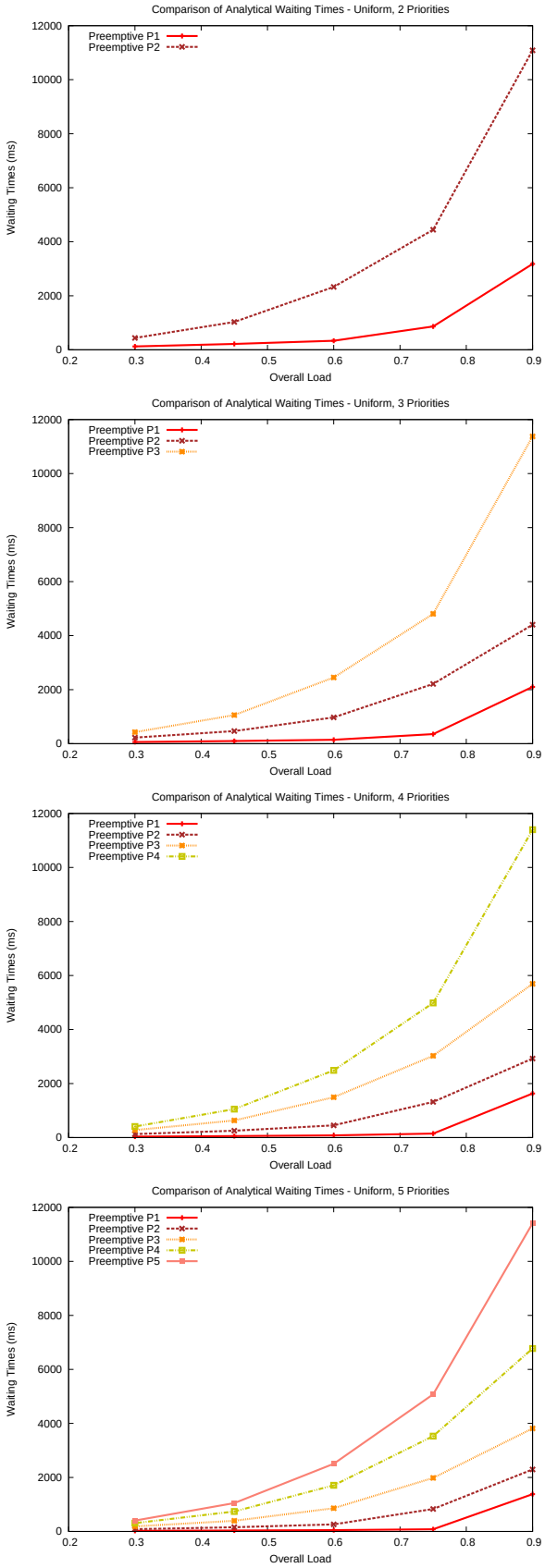

Figure 6.4: Analytical Results - Uniformly Distributed Service Times

	2 Priorities			3 Priorities			
Load	P1	P2	Overall	P1	P2	P3	Overall
0.3	125.104	436.729	169.335	59.039	223.425	424.866	106.527
0.45	214.464	1029.76	331.685	97.0203	466.997	1058.17	210.917
0.6	333.61	2327.19	619.325	143.141	972.925	2450.24	406.316
0.75	861.652	4442.95	1373.93	354.148	2211.5	4807.55	906.113
0.9	3177.39	11087.2	4307.37	2101.32	4405.02	11376	2981.27

Table 6.3: Analytical Results - Uniformly Distributed Service Times

Load	P1	P2	P3	P4	Overall
0.3	34.1952	129.461	276.767	409.01	78.4277
0.45	55.0459	252.051	631.519	1051.94	158.155
0.6	79.1888	451.273	1492.4	2486.77	311.752
0.75	147.913	1320.77	3027.1	4985.03	696.086
0.9	1631.16	2924.13	5693	11398.4	2410.48

Table 6.4: Analytical Results - 4 Priorities - Uniformly Distributed Service Times

Load	P1	P2	P3	P4	P5	Overall
0.3	22.3253	83.7994	186.012	308.476	401.435	63.7025
0.45	35.4221	154.99	395.207	740.459	1050.01	130.204
0.6	50.156	260.815	859.015	1708.48	2506	258.034
0.75	80.7494	829.822	1982.04	3529.36	5077.9	588.488
0.9	1381.02	2296.32	3812.73	6774.68	11410.4	2110.85

Table 6.5: Analytical Results - 5 Priorities - Uniformly Distributed Service Times

6.8.4 Distribution Independence Evaluation

Modelling a system as an $M/G/1$ queue allows the performance model to support arbitrary service times. The expectation is the system exhibits similar behaviour for any type of service time distribution. As such the proposed model is evaluated next for exponentially distributed service times. Figure 6.5 illustrates the estimates arrived at, using the model.

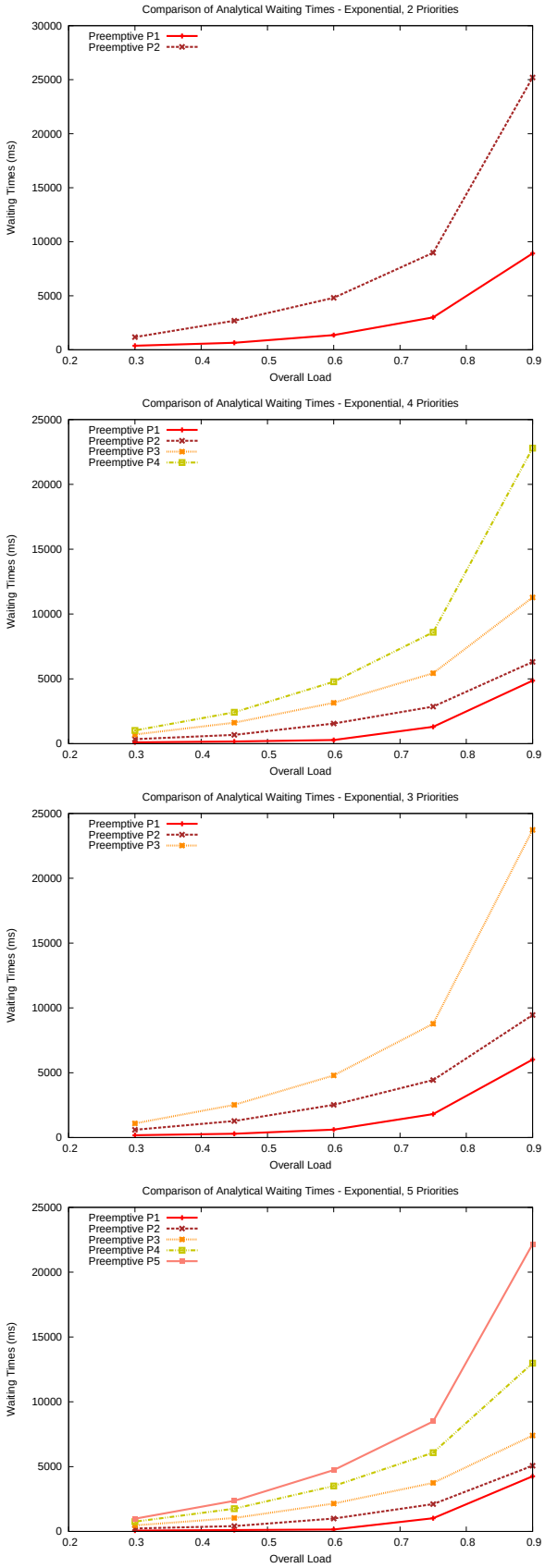

Figure 6.5: Analytical Results - Exponentially Distributed Service Times

As depicted in Figure 6.5, the higher priority classes (with shorter deadlines) having lower waiting times compared to the lower priority classes (with longer deadlines). Moreover, it can be observed that lower priority classes are estimated to have longer waiting times and be penalised more as the load increase. Comparing these trends with Figure 6.4, it can be concluded that the model predicts similar patterns for both service time distributions, thereby confirming the validity of the model.

6.8.5 Analytical vs. Simulation Results Comparison

Next the proposed model is evaluated for its accuracy of estimation of a real priority based preemptive scheduling system with $M/G/1/./EDF$ properties. For this, we compare the analytical results we obtained from value substitution, with the results obtained from simulation runs using Omnet++. Figures 6.6, 6.7, 6.8 and 6.9 contain comparisons of waiting times for scenarios from 2 - 5 priority classes. Both analytical and simulation results exhibit the same trends in waiting times for each priority class, which confirms that the model is indeed valid for any service time distribution.

With the priority based model, the goal is to favour higher priority requests thereby reducing their waiting times for processing resources. As the proposed model follows the conservation laws for a priority based preemptive scheduling system [Bolch et al., 2006; Kleinrock, 1976], work is neither created nor lost. Accordingly, favouring higher priority classes end up increasing the waiting times of lower priority classes. This is clearly visible from both analytical and simulation results obtained.

Considering the values obtained from the two, it can clearly be seen that the difference between analytical and simulation results are smaller in lower load conditions and the difference increases over higher loads. Simulation results suggest that lower priority classes are penalised less than the estimate given by the model. This is noticed in the actual waiting times in Table 6.6 for 3 priority classes and follows a similar trend with higher number priorities.

	Analytical				Simulation			
Load	P1	P2	P3	Overall	P1	P2	P3	Overall
0.3	59.039	223.425	424.866	106.527	74.2711	234.884	358.496	116.985
0.45	97.0203	466.997	1058.17	210.917	169.644	509.101	767.608	260.318
0.6	143.141	972.925	2450.24	406.316	429.908	1026.92	1546.4	593.317
0.75	354.148	2211.5	4807.55	906.113	1193.14	2120.22	3029.9	1452.07
0.9	2101.32	4405.02	11376	2981.27	5077.55	6581.77	8135.78	5494.81

Table 6.6: Waiting Times - 3 Priority Classes

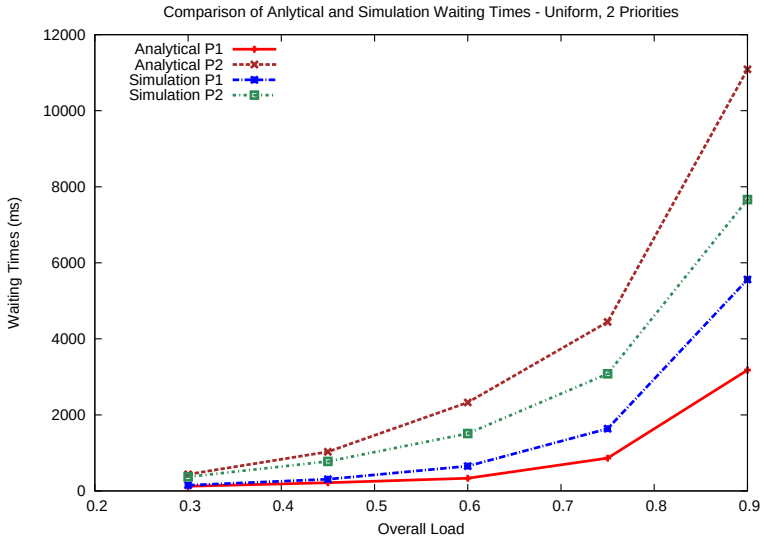

Figure 6.6: Comparison of Analytical and Simulation Results - 2 Priority Classes

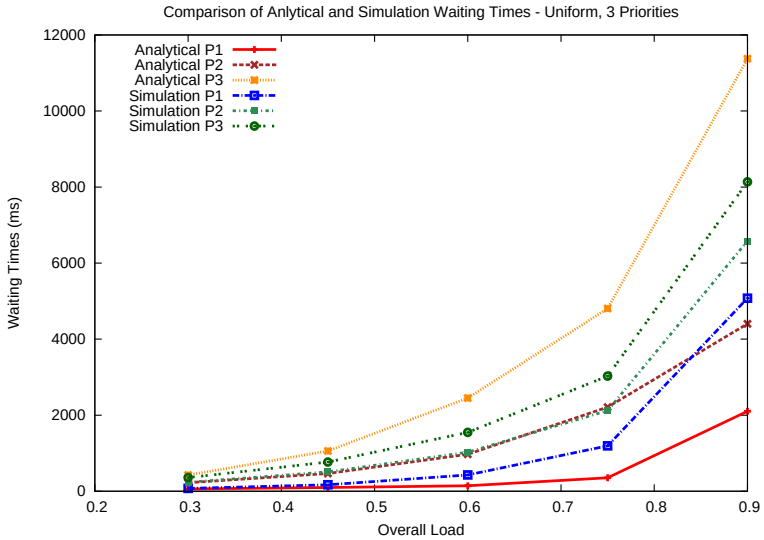

Figure 6.7: Comparison of Analytical and Simulation Results - 3 Priority Classes

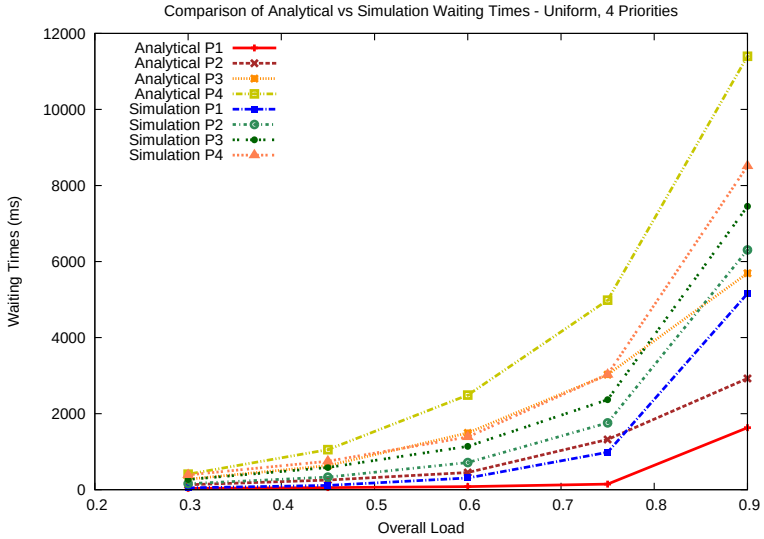

Figure 6.8: Comparison of Analytical and Simulation Results - 4 Priority Classes

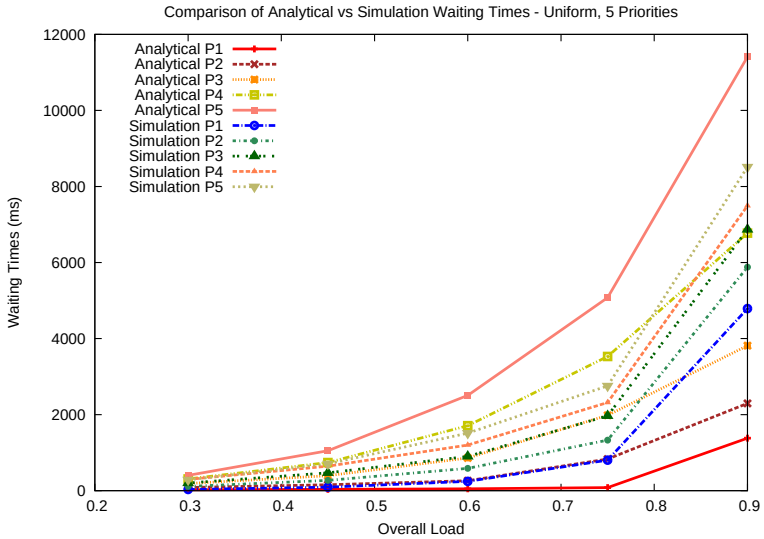

Figure 6.9: Comparison of Analytical and Simulation Results - 5 Priority Classes

6.8.6 Comparison with Non-Preemptive $M/G/1/./EDF$ system

The theoretical proof of the proposed model was based on the work of [Chen and Decreusefond, 1996], in which a model for estimation of waiting times for a Non-preemptive $M/G/1/./EDF$ system is discussed in detail. While modelling a preemptive $M/G/1/./EDF$ is quite

complex due to the preemptions, using such a model would ensure that higher priority classes receive better waiting times. Moreover, a preemptive scheduling system using EDF as the algorithm is a better representation of the middleware supporting predictability that we have discussed in previous chapters.

Figures 6.10 - 6.13 contains the comparison of analytical results obtained from the proposed model and the model discussed in [Chen and Decreusefond, 1996].

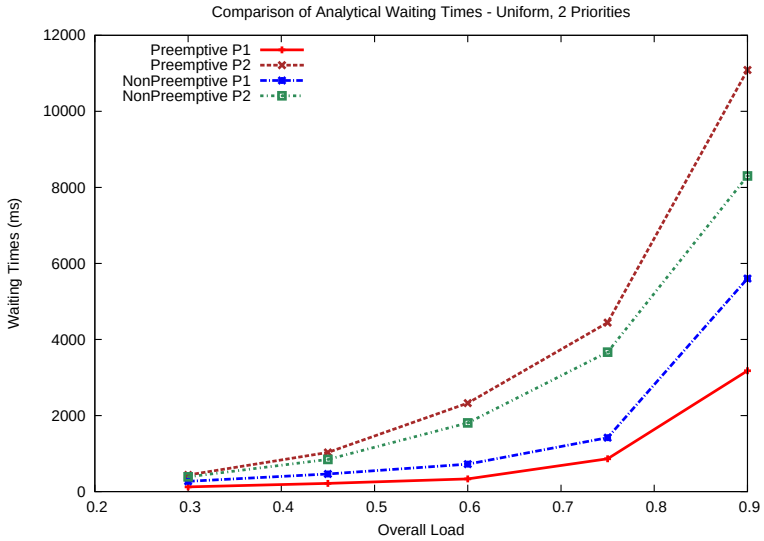

Figure 6.10: Comparison of Analytical Results - 2 Priority Classes

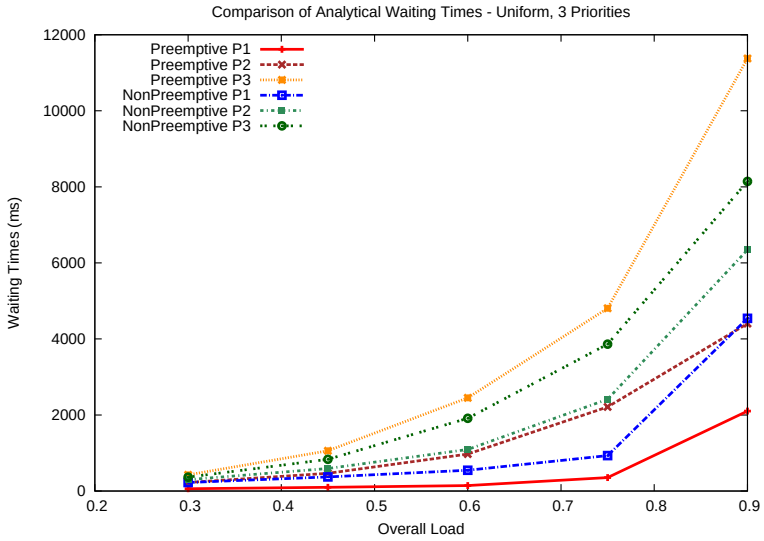

Figure 6.11: Comparison of Analytical Results - 3 Priority Classes

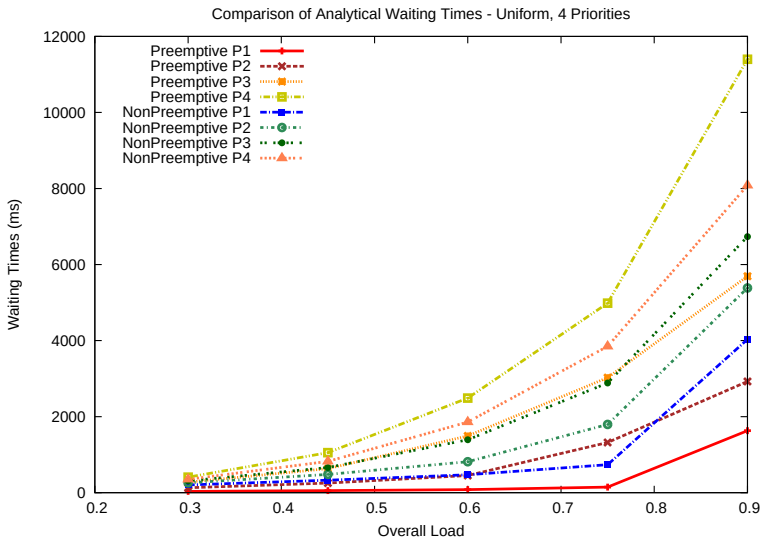

Figure 6.12: Comparison of Analytical Results - 4 Priority Classes

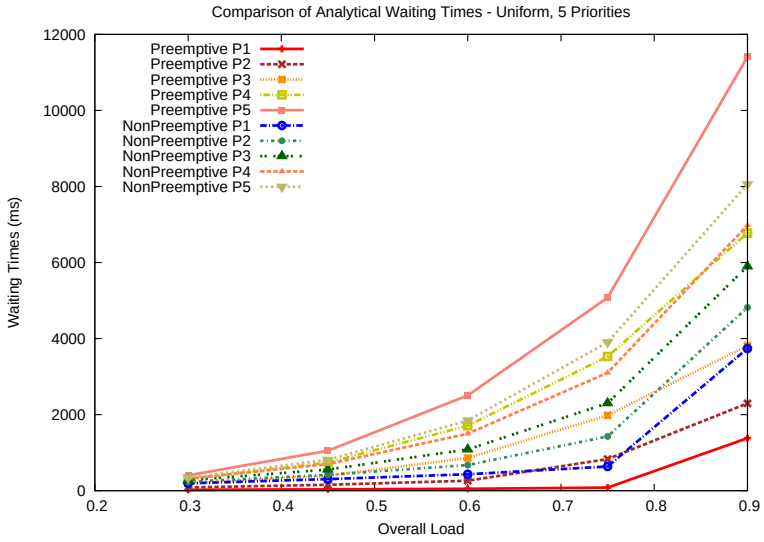

Figure 6.13: Comparison of Analytical Results - 5 Priority Classes

As expected, the proposed model favours the higher priority classes and estimates lower waiting times for those classes compared to the non-preemptive model. Estimates from the non-preemptive model results in shorter waiting times for lower priority classes compared to the preemptive estimates. Moreover, the differences in waiting times between priority classes for a given load tends to be more uniform with the non-preemptive

estimates, while in the preemptive scenario the differences get larger as the priority becomes lower.

Non preemptive scheduling ensures that a task in execution is never interrupted by newly arriving tasks, irrespective of their priority. The request queue in such a system only grows with request that are yet to be executed. The queue in a preemptive scheduling system contains tasks that are yet to be executed as well as tasks that have been partially executed and were preempted by incoming higher priority tasks. This results in non-preemptive systems having comparatively better waiting times for lower priority requests. Conversely, preemptive systems favour high priority requests by letting them use the processing resources at the earliest possible by suspending the execution of lower priority requests and resuming them later when there are no higher priority requests. Hence, the proposed model returns better waiting times for higher priority requests.

6.8.7 Comparison with simple $M/G/1$ priority systems

Next we compare the analytical results obtained from the proposed model with that of a simple priority based $M/G/1$ queueing system. Herein, the priority is enforced as a static property where they are assigned *apriori* and enforced in a manner where a higher priority request will always preempt a lower priority request. As a result the default behaviour would be for a request with a higher priority to always preempt a request of a lower priority, upon its arrival at the system. We also include the non-preemptive EDF based model by [Chen and Decreusefond, 1996] as well as a simple $M/G/1$ queue for this comparison.

Figures 6.14 - 6.17 contain the comparison of waiting times for each priority class for different priority scenarios with 2 - 5 priorities. In each scenario the simple $M/G/1$ models result in lower waiting times than the two EDF based models for higher priority classes. This observation is valid for all load conditions. Moreover, the two simple $M/G/1$ models exhibit an almost linear increase in waiting times with system load for higher priority classes, whereas the two EDF based models show an exponential type increase.

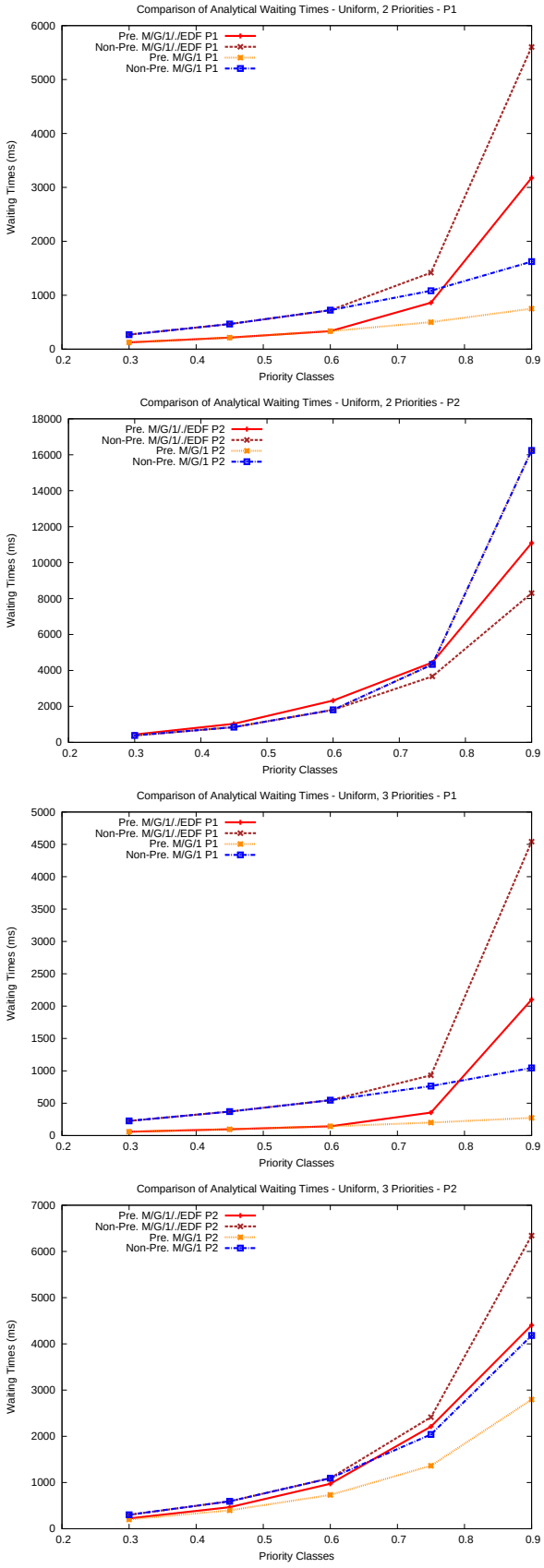

Figure 6.14: Comparison with simple $M/G/1$ priority systems

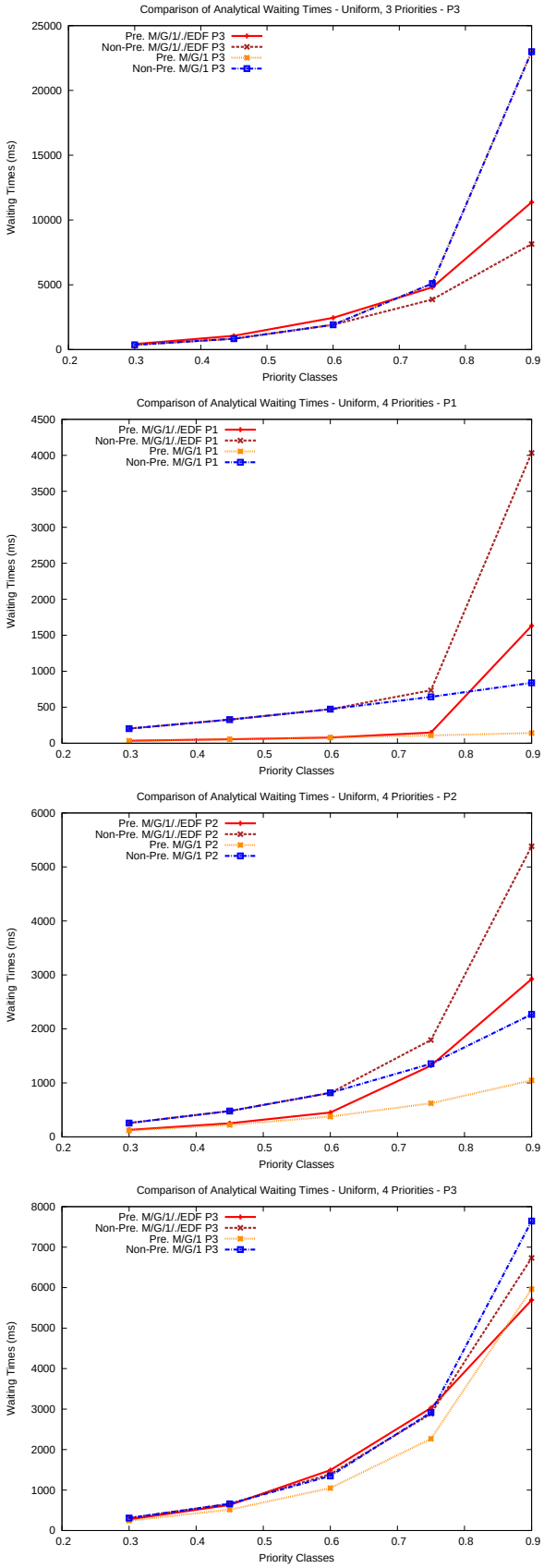

Figure 6.15: Comparison with simple $M/G/1$ priority systems

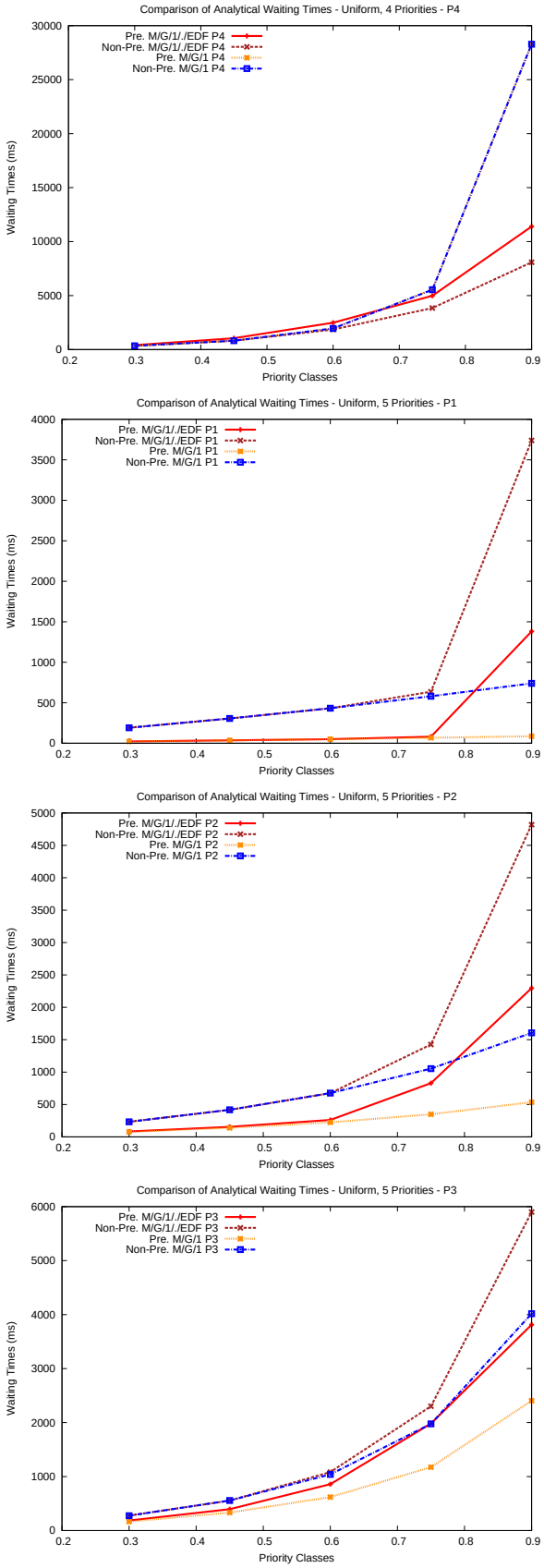

Figure 6.16: Comparison with simple $M/G/1$ priority systems

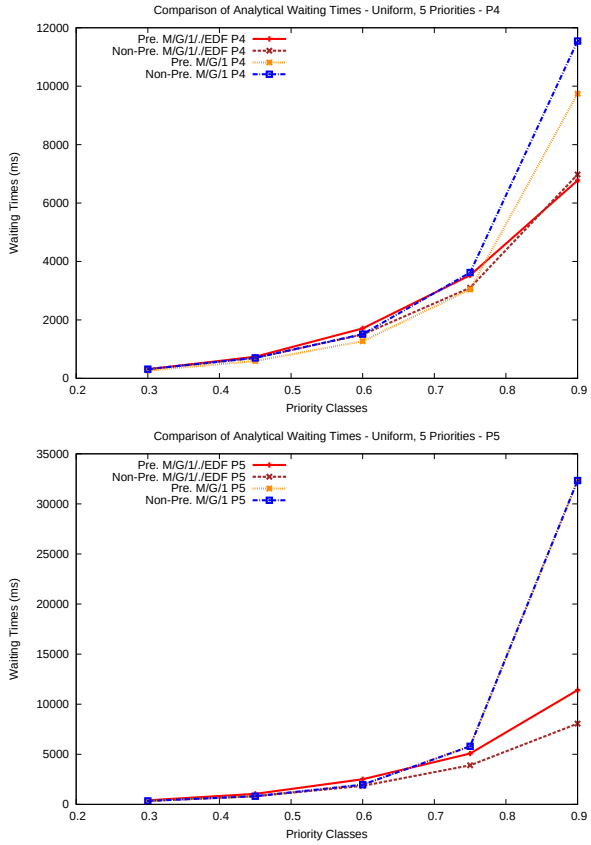

Figure 6.17: Comparison with simple $M/G/1$ priority systems

When lesser priority classes are considered, this effect is reversed with simple $M/G/1$ models resulting in higher waiting times for lower priority classes than the EDF based models. It can be observed that the simple models result in waiting times that are a couple of times than the EDF models for the lowest priority in the system and the difference becomes larger with the number of priorities in the system. Moreover, note that both preemptive and non-preemptive simple $M/G/1$ systems record the same mean waiting time for the lowest priority class. This phenomenon is due to the waiting times the lowest priority class experience and the mean residual service time calculation in each model being calculated the same way for the lowest priority.

The fixed priorities used in simple $M/G/1$ systems result in comparatively shorter waiting times for higher priority requests. Herein, a request from a higher priority class can preempt a request from a lower priority class any time in its execution. Similarly, this also leads to lower priority requests being penalised heavily especially in high load

conditions due to the larger number of higher priority requests.

With the absolute deadlines deciding the priority of a request at a given time in the EDF based scenarios, there are instances where a request from a lower priority class in execution will have the higher priority at a given time than a newly arrived request from a higher priority class due to its absolute deadline being earlier than that of the newly arrived. While this behaviour results in slightly longer waiting times for higher priority classes than simple $M/G/1$ models, this method prevents lower priorities from starvation and tries to achieve a better balance between classes. However, the intended priority levels and service differentiation is maintained as intended. Such a balancing techniques that reduce starvation of lower priority requests are considered as important strategies and favoured in priority based systems.

6.8.8 Comparisons with Other Algorithms

Next, we compare the EDF based preemptive scheduling algorithm implemented with the characteristics of an $M/G/1$ queue with First-Come-First-Served (FCFS), Round-Robin (RR) and Non-Preemptive Priority Ordered (PRO), three popular algorithms that are widely used in distributed and web systems. Herein, each algorithm is implemented using Omnet++ and are exposed to different simulated traffic conditions. We measure the waiting times for each priority class and the overall waiting time of the system, over five, low to high system load conditions. The FCFS system queues requests in the order of their arrival and does not consider the deadline requirement in any part of the scheduling process. Moreover, it does not differentiate between the priority classes in request processing or execution. The RR system uses separate queues for each priority class and selects requests among them for execution in a round-robin manner. The PRO system is non-preemptive and enforces priorities in a static way where a class designated to be a higher priority is always given the preference over another.

Figures 6.18 to 6.22 contain the comparisons for each priority class as well as the overall waiting time comparison between the systems. The PRO algorithm achieves the lowest waiting times for the highest priorities in every configuration.

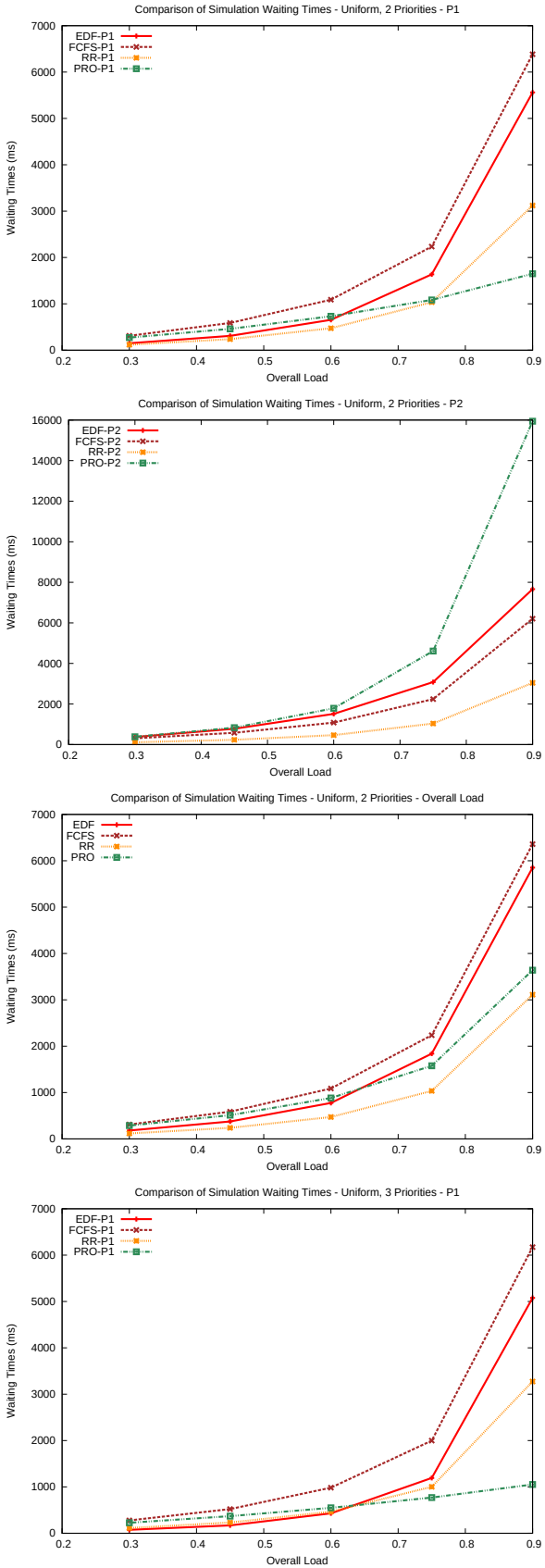

Figure 6.18: Comparison of Simulation Results - All Algorithms

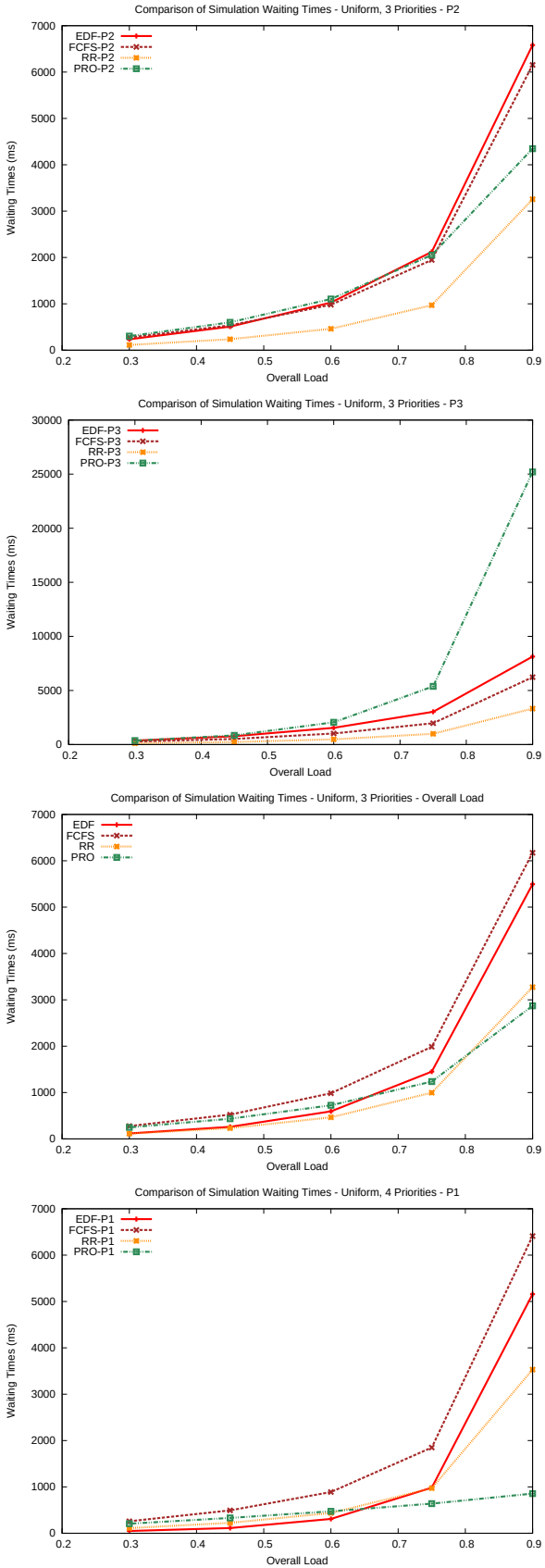

Figure 6.19: Comparison of Simulation Results - All Algorithms

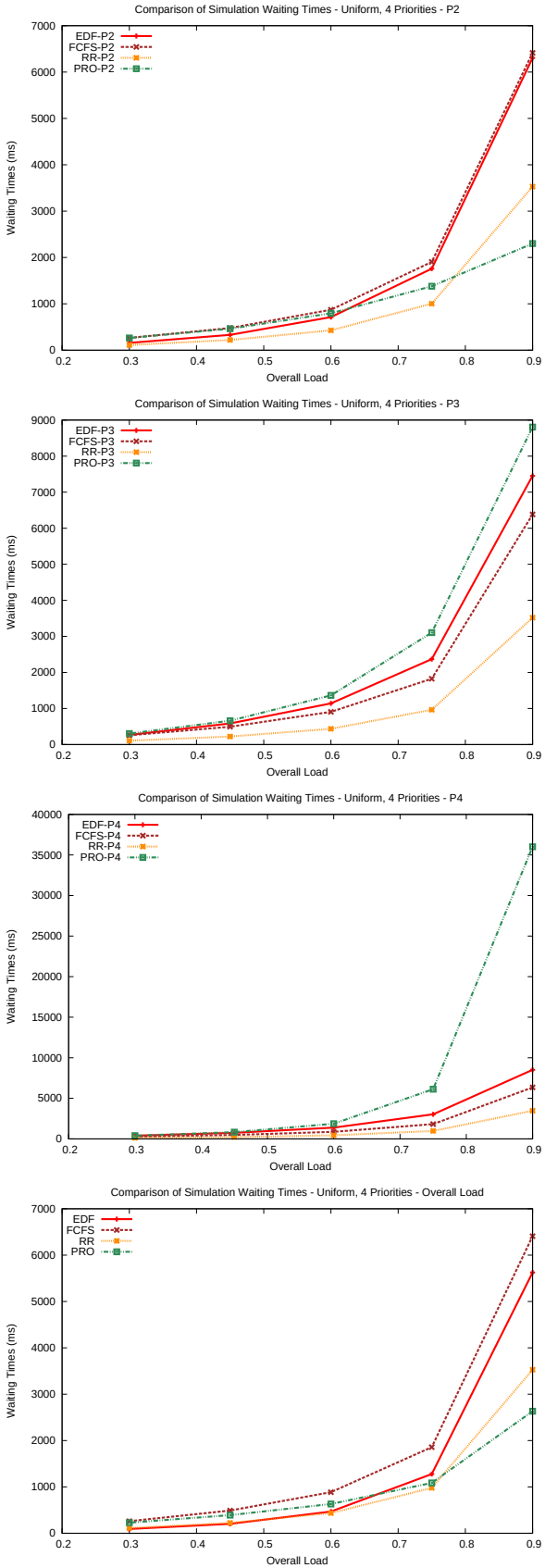

Figure 6.20: Comparison of Simulation Results - All Algorithms

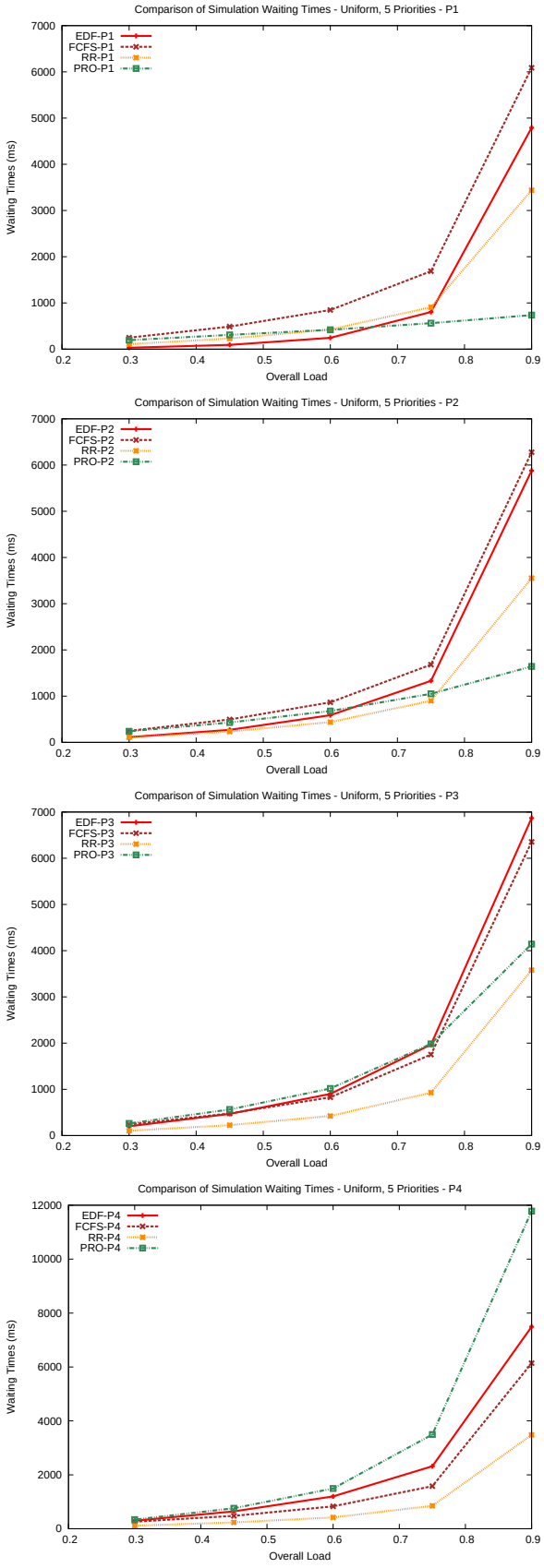

Figure 6.21: Comparison of Simulation Results - All Algorithms

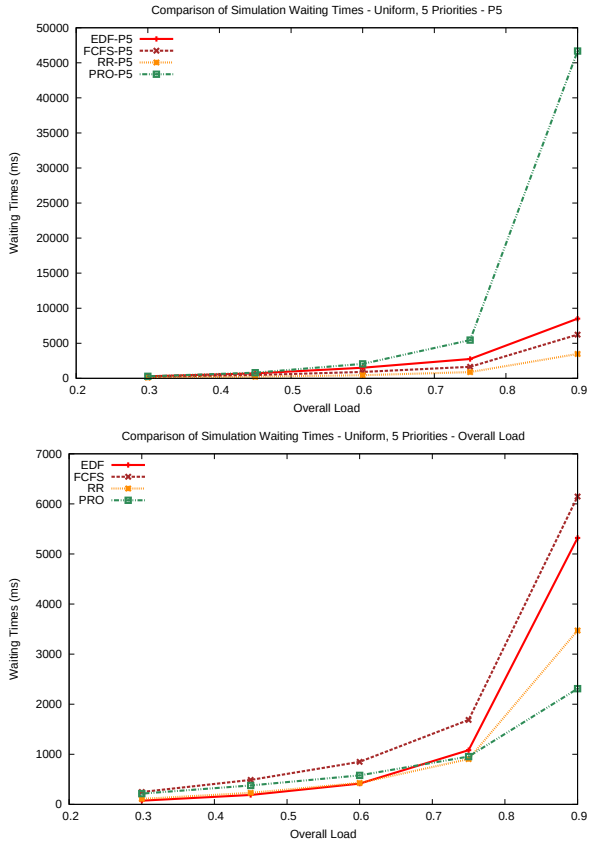

Figure 6.22: Comparison of Simulation Results - All Algorithms

As the load increases the waiting times recorded by EDF, FCFS and RR algorithms show an exponential increase, while the PRO algorithm records a near linear increase in waiting times as predicted by the analytical results. The FCFS system records the longest waiting time for the highest priority of all the algorithms. With lower priority classes, FCFS and RR systems record comparatively shorter waiting times. The PRO system records the highest waiting times out of all algorithms. Another observation is that the waiting times recorded by FCFS and RR algorithms are similar across the different priority classes. The EDF algorithm neither records the shortest nor the longest waiting times for any of the priority classes and exhibits balanced behaviour throughout the priorities. However, the intended differentiation of processing is clearly visible, where higher priority classes have preference, yet without over starving the lower priority classes.

The FCFS and RR algorithms, in their functionality treat all classes with equal priority.

This behaviour is confirmed by the similar mean waiting times they record across all priority classes. The PRO algorithm is designed to favour higher priority requests. This is confirmed by it achieving the shortest waiting times for higher priority requests and longest waiting times for the lower priority requests out of all the algorithms. Its non-preemptive nature favours higher priority requests than the EDF based system, while penalising the lower priority classes more. In terms of the waiting times recorded by each algorithm, RR algorithm seems to be better at recording lower waiting times than the others. However, we associate a soft processing deadline for each request which is an indicator of the appropriate time the processing would be expected to be completed within. Although some of these deadlines would be missed in processing and such requests are still deemed to be valid, the deadline miss rate would also be an indicator of the performance of an algorithm. Tables 6.7 and 6.8 contains miss rates recorded by the four algorithms for 3 priority classes and 4 priority classes respectively.

	Preemptive $M/G/1/./EDF$				Non-Preemptive PRO			
Load	P1	P2	P3	Overall	P1	P2	P3	Overall
0.3	0.44	1.69	3.01	0.79	4.27	2.48	1.80	3.83
0.45	2.32	6.25	11.23	3.47	7.39	6.76	7.48	7.28
0.6	8.81	17.32	26.05	11.21	12.26	15.81	19.22	13.25
0.75	24.36	36.17	48.15	27.68	32.50	36.19	38.24	33.45
0.9	59.03	69.28	77.95	61.77	63.95	69.17	70.28	65.17
	Round-Robin				FCFS			
Load	P1	P2	P3	Overall	P1	P2	P3	Overall
0.3	3.00	4.89	5.38	3.45	6.29	1.79	0.56	5.21
0.45	8.22	14.42	18.43	9.84	13.08	5.07	1.88	11.09
0.6	18.20	30.26	36.46	21.28	25.04	13.13	7.74	22.04
0.75	35.66	51.46	59.54	39.69	43.49	28.71	18.89	39.61
0.9	66.91	79.40	85.15	70.01	72.32	62.85	53.89	69.74

Table 6.7: Miss Rates - All Algorithms - 3 Priorities

	Preemptive $M/G/1/./EDF$					Non-Preemptive PRO				
Load	P1	P2	P3	P4	Overall	P1	P2	P3	P4	Overall
0.3	0.51	1.61	2.92	5.16	1.00	6.60	3.98	2.61	2.62	5.73
0.45	2.17	5.02	9.47	14.64	3.55	11.74	8.64	7.71	7.06	10.76
0.6	8.01	14.67	22.20	27.90	10.73	20.28	18.69	18.39	15.75	19.74
0.75	24.27	35.39	44.77	53.57	28.51	36.42	39.29	38.15	39.21	37.14
0.9	62.13	70.19	76.88	80.63	65.17	68.82	71.71	72.69	72.89	69.74
	Round-Robin					FCFS				
Load	P1	P2	P3	P4	Overall	P1	P2	P3	P4	Overall
0.3	4.44	5.42	6.56	7.63	4.85	8.58	3.74	1.39	0.40	6.98
0.45	10.31	13.16	18.83	19.12	11.68	16.82	8.13	3.46	1.67	13.88
0.6	19.98	27.54	34.84	36.91	22.83	28.92	16.90	10.01	6.43	24.83
0.75	38.64	50.08	58.38	62.50	42.76	48.20	36.14	25.74	18.72	43.64
0.9	72.11	79.38	85.27	85.89	74.77	77.69	70.24	60.97	56.13	74.52

Table 6.8: Miss Rates - All Algorithms - 4 Priorities

From the miss rates recorded, it is clearly visible that the EDF based system achieves the lowest miss rates in all priority levels. EDF, being an algorithm that considers the processing deadline as its main parameter for making request selections or scheduling decisions, is by design optimised to achieve such processing deadlines and such behaviour is expected. Out of the three algorithms that do not consider a deadline, PRO achieves the lowest miss rates. Such behaviour is yet again expected as it favours higher priority requests, in this case the majority of requests in the system and having the shorter deadlines compared to the other classes.

As FCFS executes requests in the order of their arrivals, comparatively more higher priority requests will still be executed due to their shorter inter-arrival times. However, a lower priority request with a long processing time will still be executed when its at the head of the queue. This will lead to higher priority requests queueing behind and waiting a long time till the lower priority request completes execution. Thus, it leads to majority of the higher priority requests missing their deadlines. However, this non-differentiation will lead to a lower miss rate for the lower priority requests due to their comparatively smaller number in the system. As the RR system uses different queues for the priority classes, there is no chance of requests with mixed priorities to queue behind each other. However, the round-robin selection of requests among these queues ensures that requests in each queue has an equal chance of being the next request to get at the processor. Given this condition, the miss rates from the RR system follow the pattern of being smaller for higher priority classes and larger for lower priority classes. However, as there are separate queues being used, the waiting times recorded are comparatively longer for higher priority requests (second only to FCFS). Due to the similar reasons, arrival rate differences between the priority classes are directly reflected in the waiting times recorded by the RR system.

6.8.9 Discussion

The evaluation of the proposed priority based preemptive $M/G/1/./EDF$ model was evaluated in many categories. The theoretical evaluation we provided confirms that the proposed model could be used to derive a set of equations that solve in an iterative manner to estimate the waiting times of the intended system. The mean waiting times obtained through substitution as part of the evaluation confirms our understanding and expected behaviour of such a system. Therefore, we could conclude that the proposed

model is valid and can be used to estimate the mean waiting times of the target system. The model was further evaluated using configurations with different priority levels and the evaluation confirmed that irrespective of the number of priorities present in the system, the resultant mean waiting times follow a similar pattern thus far confirming the validity of the model for any number of priority classes.

The proposed model considers the method of scheduling to be independent of the service times used. To evaluate this aspect, we used a configuration with exponentially distributed service times on the model and measured the mean waiting times for different load conditions and priority class counts. However, the mean waiting times obtained followed a similar pattern for both uniformly and exponentially distributed service times, thereby confirming that the model certainly supports any service time distribution.

Thereafter we compared the analytical results obtained with the actual waiting times recorded by a simulated system under various traffic and load conditions. These evaluations were conducted with the view of measuring the accuracy of the waiting times given by the model. In these evaluations, it was observed that there was a difference between the analytical and simulation results and the difference becomes significant with increasing system load. Moreover, the increase of waiting times with load, for each priority class is much higher in high load conditions. Note that the proposed model is a mathematical approximation of the actual system and it provides estimates based on statistical parameters. Therefore, differences in the estimates and the actual times recorded, are to be expected.

In the next set of evaluations we compared the proposed model with the non-preemptive $M/G/1/./EDF$ model we based our theoretical proof on. These were conducted to justify the performance gain we achieve by using preemptive scheduling instead of being non-preemptive. Our comparisons revealed that a preemptive model favours higher priority requests, thereby achieving comparatively shorter waiting times than the non-preemptive model. The ability for a higher priority request to immediately seize the processing resources upon its arrival at the system and by preemption even when a lower priority task is in execution, confirms this behaviour. However, due to the conservative nature of the proposed model, lower priority requests experience comparatively longer waiting times than in a non-preemptive model.

Thereafter, a detailed comparison of the proposed model with two preemptive and non-

preemptive $M/G/1$ priority systems was carried out. The comparison also included the non-preemptive EDF based model we based our proof on. The simple $M/G/1$ models favour higher priority requests more than the EDF model due to their static priority enforcements. With the deadline based scheduling, although the general priorities of request classes are decided at design time of a system, a request of a lower priority class could at a given time be the highest priority request in the system for having the earliest deadline of all requests present. While this may seem as if having a negative impact on higher priority classes, on the contrary it results in a better balance in the system together with the lower priorities. This better waiting times recorded by the EDF models for the lower priorities, confirms this phenomenon. Herein, the better balance we thrive to achieve is favouring the higher priority requests while not over starving requests of the lower priority classes.

To evaluate the performance of the EDF based scheduling implemented with a $M/G/1/$ queue on a system with soft deadlines, we compared it with FCFS, RR and Non-preemptive PRO algorithms. The non-differentiating nature of the FCFS and RR algorithms, were demonstrated in the similar waiting times they achieve for all priority classes. The PRO algorithm follows the priority scheme by design, thus favours the higher priority request classes resulting the better waiting times for higher priority requests and offers the best waiting times for high load conditions for such requests. However, given the static priority model it follows and its non-preemptive nature, it records the longest waiting times for the lower priority classes and penalises them more on higher load conditions. While the $M/G/1/./EDF$ system also follows the priority model, it enforced them dynamically based on absolute deadlines where a request from a lower priority class can be the request with the earliest deadline thereby having the highest priority at a given time. Such behaviour favours the higher priority classes with comparatively shorter deadlines while giving the lower priority classes a chance and preventing them from over starvation. Moreover, the deadline miss rates recorded by each algorithm indicates that algorithms such as FCFS and RR result in a high percentage of deadline misses, despite the lower waiting times they record at times. From the results, we could conclude that explicitly scheduling based on a deadline will give systems a better chance of meeting their processing and QoS guarantees.

For a system that uses a priority model and differentiated processing, such behaviour could be considered optimal depending on the requirements. In web services and cloud middleware that are increasingly moving towards multi-tenancy, it is a must to achieve

the right balance between the different tenants, their clients and request types and. From the evaluations presented, it can be concluded that deadline based scheduling could be used in them to achieve the right balance between such parameters. As shown by our model, the service time independent nature of the scheduling technique makes it usable with any type of requests. Moreover, the use of deadlines enable the modifying the priority levels of requests on-the-fly, thereby enabling more control on how requests are executed at runtime. Such features will enable service providers with zero downtime changes to their system's scheduling discipline.

6.8.10 Difference Between Analytical and Simulation Results

A possibility for the discrepancy of analytical and simulation results is in the estimation of mean delay incurred by tasks in execution at an arrival, given by equation 6.6.7. In the estimation, the calculation was based on the probability of finding a task belonging to class 1 to i in execution. This is considered due to classes $i+1$ to N having larger deadline offsets, being treated as low priority. Although this condition holds true theoretically, a newly arrived task from stream i in an actual system using EDF scheduling, could find a task belonging to class $j = (i+1)..N$ in execution and not preempt it due to absolute deadline considerations in preemptive tasks. A task belonging to stream $j = (i+1)..N$ that arrives at least $D_{j,i}$ time units prior to the aforementioned request from stream i will be in execution at its arrival and will not be preempted by it.

An observation made in Figures 6.6 to 6.9, is the difference between the two results increasing with the system load. The differences become quite significant in the highest load conditions. Upon analysing the simulations it was found that many tasks miss their deadlines in the high load conditions. This is clearly visible from the miss rates for the simulations that were presented in Tables 6.7 and 6.8 under Preemptive $M/G/1/./EDF$. Deadline misses lead to execution of requests getting longer, thereby having an impact on the mean service completion time $\overline{C_i}$ and in turn the mean delay incurred by the requests in execution $\overline{W_0^i}$. For instance, a stream i request arriving at the system may find a task from stream $j = (i+1)..N$ already in execution. The newly arrived stream i request may not be able to preempt the stream j request due to it having an earlier deadline despite having missed it. Although the proposed model has this as a parameter, it does not contain a representation for deadline misses or the impact it may have on $\overline{C_i}$ and on $\overline{W_0^i}$.

Each of these observations contributes to the discrepancy between analytical and simulation results. Given the complex nature of the system, it is difficult to modify the model and consolidate these differences to get the values closer. However, both group of results display the same characteristics despite the differences in actual values. Therefore, we could conclude that the model presented is an acceptable approximation of a $M/G/1/./EDF$ system.

6.9 Summary

In this chapter we presented a performance model for web services middleware that uses EDF scheduling. The proposed model was based on queueing theory and modelled the system as a multi-priority based preemptive resume $M/G/1/./EDF$ queue where the priority ordering is governed by the execution deadlines. The performance metric of concern was the mean waiting time experienced by requests belong to each priority class.

We provided analytical proof that the proposed model represents such a system with reasonable accuracy and evaluated the model against non-preemptive representations of similar queues and several other algorithms for various load conditions and priority configurations. Next, the accuracy of the proposed model was evaluated by comparing the analytical results obtained, with waiting times recorded through simulation of actual traffic conditions. Both sets of results exhibit the same characteristics of favouring higher priority request classes and resulting in higher waiting times for lower priority classes. While there are minor differences between analytical and simulation results, we provided reasoning for such a discrepancy and concluded that the approximations made by the model were valid given the reasons for the difference. When compared to other algorithms, EDF based scheduling seem to follow the intended behaviour of favouring the higher priority request classes whilst preventing the lower priority requests being over starved. Therefore, we concluded that EDF would be a better choice for systems where differentiated request processing between multiple classes is required, yet a balanced approach where lower priority requests are prevented from over-starvation, is considered important. Moreover, being a solution that is valid for any service time distribution, the model is valid for any system that uses EDF scheduling, not being limited to web services middleware.

Chapter 7

Discussion and Conclusion

This thesis investigated an important aspect of web services performance, namely predictability of execution. Existing web services middleware fail to achieve predictable execution times due to their inherent designs and optimisations. Our work was motivated by the growth of web services usage and the growing popularity of cloud computing, the new computing paradigm web services have made possible. With everything being offered as a service on the Internet, execution time QoS mandates increased attention.

Many of the existing work in execution time QoS either make the assumption that the underlying middleware would ensure the QoS levels are met, or they manage to only achieve some level of differentiated request processing. This holds true for both standalone web services middleware and cluster based web services deployments. However, most of the existing solutions do not consider predictability of execution or fail to complete the execution of a service within a perceived deadline in a repeatable and consistent manner. A few of them that support a processing deadline achieve this in closed systems where properties of tasks are known at design time of the system.

At a high level, the proposed research shows that consistent execution of web service requests within a given deadline in open systems where task properties are unknown, can only be achieved by satisfying three important factors. Firstly, requests must be explicitly scheduled for execution based on their deadline. Secondly, requests must be selected for execution based on the ability to meet the deadline requirement and the possibility to delay their execution without missing the deadline. Finally, some method of

differentiation must be employed to have control over their execution. Moreover, their execution must be supported by proper development platforms, libraries and operating systems that ensure predictable execution times in all levels of the software stack.

Satisfying these factors, Chapter 3 addressed the problem in its simplest form, of achieving predictability of execution in stand-alone web services middleware. We introduced the notion of a processing deadline, presented means of admission control and request scheduling based on real-time scheduling principles. In Chapter 4 we extended our solution to a cluster based deployment of web services where the techniques presented in Chapter 3 was supported by four dispatching algorithms that ensure the deadline of a request can be met at the executor it is assigned to.

Chapter 5 presented the practical aspects of building web services middleware (or enhancing the ones available) to support processing deadlines and ensure they will be met consistently. The software engineering techniques, designs, algorithms and tools presented are generic enough to be used with any product in use. Moreover, we presented concise guidelines to help in identifying predictability features existing middleware already possess and what enhancements are required. In chapter 6 we present an analytical model for a system using deadline based scheduling and derive advanced performance metrics that enable us to analyse the effects of such deadline based scheduling on the overall performance of the system.

7.1 Summary of Contributions

In this section we summarise the contributions made in relation to the research questions specified in section 1.2.

How can predictability of execution be achieved in stand-alone web services middleware?

In addressing this question we presented a mathematical model and a supporting algorithm based on real-time scheduling principles, for an admission control mechanism that selects requests for execution based on their laxity. It not only guaranteed the deadline of a request upon selection but also ensured that deadlines of already accepted requests would not be compromised. The selected requests were scheduled for execution using earliest deadline first scheduling principle. Real-time scheduling principles are typi-

cally used at design time in closed systems to schedule tasks with known properties. The uniqueness of the proposed solution was the use of such scheduling techniques at run-time in an open system where task properties are unknown.

How can predictability of execution be achieved in cluster based web service deployments?

The contributions made for the second question were four request dispatching algorithms that work together with the techniques introduced into stand-alone web services middleware (hosted in each executor on the cluster). Each algorithm maps a request to an executor in a different way and takes the additional step of ensuring that the processing deadline of the request can be met with the selected executor. Two of the algorithms perform dispatching in a content-blind manner and the remaining two were content-aware dispatching algorithms. Three of these algorithms ensure the schedulability of a request prior to being dispatched while the laxity-based algorithm considers the laxity property even for the matching of a request to an executor. The laxity of a request being considered in two steps gives the best chance for a request to achieve its processing deadline.

How can web services middleware products be engineered to have predictable execution times?

Addressing this question, the contribution made was the practical aspects of building middleware products to support predictability of execution. The concepts and algorithms introduced as contributions to the first two questions needed to be implemented properly, supported by the middleware. For this purpose, we presented software engineering techniques, designs, algorithms and tools that can be used in the development process. We also discussed the possible challenges faced in the development process and how to overcome them. Moreover, we provided a set of guidelines that can be used as a reference in the process, as well as to be used in identifying the capabilities of existing middleware and enhancements they require.

How can advanced performance metrics be obtained from deadline based scheduling systems?

The contribution made in addressing this question is an analytical model of the system as a preemptive $M/G/1$ queue using EDF scheduling policy. Its uniqueness stems from the fact that this is the first time an analytical model has been defined for this type of queue being used with EDF policy in a preemptive scheduling system. With this performance model, we were able to approximate the waiting times for a multiple priority system with any number of priority classes. The uniqueness of the solution lies in the fact that while priorities of the different classes are decided beforehand, it is enforced dynamically at runtime, as the priority is ultimately decided on the absolute deadline of a request. This phenomenon enabled the performance model to achieve a more balanced performance where higher priority classes get more preference (thereby smaller waiting times), while the lower priority requests do not reach over-starvation.

7.2 Discussion

In this section we summarise the findings from the evaluations conducted and reflect on their importance for the research questions in section 1.2.

Execution time predictability in stand-alone web services middleware

We evaluated the admission control mechanism and deadline based scheduling method introduced into stand-alone web services middleware, by implementing them in Apache Axis2 middleware product. We measured the predictability gain it achieved with the enhancements made, by comparing it with an unmodified version of the product. Both systems were exposed to different traffic conditions and the resultant loads. The empirical evaluation concluded that the enhancements made to the middleware enabled it to achieve at least 96% of the requested deadlines while accepting at least 18% of the requests from the schedulability check, in very high traffic conditions. The unmodified version only manages to meet at most 36% of the deadlines (in low traffic conditions) while accepting at least 28% of the requests. The results also confirmed our claim that *best-effort* processing results in a large range of execution times due to processor sharing. Moreover, the results confirmed that the laxity based schedulability check prevents the system getting overloaded and enabled it to maintain a good throughput rate even

when faced with high system loads. While a portion of requests were rejected in the process of achieving predictability for others, the CPU utilisation levels indicated that the processor was kept fully utilised throughout the experiment, justifying the rejections or the possible overloading of the server if a higher percentage of requests were accepted.

Execution time predictability in web service clusters

The predictability gain achieved by introducing attributes of predictability into request dispatching was evaluated by comparing the performance of the algorithms with others. The comparison of the predictability gain by RT-RoundRobin against simple RoundRobin was a clear example of the predictability gain in its simplest form. RT-RoundRobin was able to meet at least 90% of the deadlines with just 2 executors while accepting at least 20% of the requests in high traffic conditions. In similar conditions, a simple round-robin algorithm resulted in 39% of accepted requests (due to the servers being overloaded and requests timing out) while only managing to meet 6.3% of the deadlines. The large range of execution times that result (due to *best-effort* processing) with simple round-robin scheduling was evident from the results.

RT-ClassBased is an example of predictability attributes being considered in an algorithm that unbalances the load. Its comparison to class-based dispatching which simply uses the request size as the deciding parameter of a class. With the predictability enhancements, RT-ClassBased was able to achieve at least 95% of the deadlines while accepting 29% of the requests, while class-based was able to achieve just 8% of the deadlines when accepting at least 52% of the requests with the most task arrivals. Therefore the predictability gain by the changes were noticeable. With RT-ClassBased, the utilisation level at each executor corresponded to the amount of work each executor was assigned with (i.e. executors assigned with large tasks were utilised more than the ones executing the smaller sized tasks, due to uniformly distributed task sizes).

RT-LaxityBased algorithm incorporates the laxity property with the decision making process of matching a request to an executor. It was designed to ensure a larger range of laxities at each server. This controlled distribution of laxities was expected to yield better results than the other algorithms as predictability based attributes are considered twice in the dispatching decision. Confirming our expectations, the empirical results indicated that RT-LaxityBased indeed achieves best deadline achievement rates of all four

algorithms. RT-Sequential takes an exhaustive approach to dispatching by checking the schedulability of a request with every executor prior to rejection. Given its exhaustive approach, it records the lowest deadline achievement rate of all four algorithms. However due to similar reasons, it achieves the second best acceptance rates out of the four algorithms.

All four of our dispatching algorithms clearly outperformed the algorithms they were compared against. This validates our attempt at incorporating predictability based decision making into the request dispatching process in a cluster. Moreover, it is clear that even a simple dispatching technique could be enhanced to consider predictability based attributes in dispatching, as we have demonstrated with RT-RoundRobin. They demonstrated that both content-blind or content-aware algorithms can be enhanced to incorporate predictability based attributes. The four algorithms have their own merits and limits, which makes them suitable for different types of request streams. As the empirical evidence suggest, predictability focused dispatching of requests results in a large range of laxities at each executor. This allows the server to accommodate the deadlines of many requests, in turn an increased request acceptance rate. The schedulability check that is part of all algorithms prevented the system from being overloaded with requests. This was evident from the throughput comparison conducted. Moreover, while being resilient to high traffic conditions, the enhanced cluster achieves comparable throughput rates with the unmodified cluster in low traffic conditions.

Advanced performance modelling of EDF scheduling in web services

The mathematical model presented for a priority based preemptive $M/G/1/./EDF$ system was intended to have better waiting times for higher priority requests without leading lower priority requests into over-starvation. The theoretical evaluation confirmed that the resultant set of equations from the proposed model can indeed be solved using an iterative process. Subsequently the analytical and simulation results obtained, confirmed our intention that the model indeed is a valid approximation of waiting times for such a system and the intended behaviour is represented in the results. Compared to a non-preemptive model for a similar queue, our preemptive model achieves better waiting times for higher priority request classes. The comparison with both preemptive and non-preemptive simple $M/G/1$ queues confirmed that EDF based scheduling is indeed a more balanced approach where higher priority requests are favoured without

over penalising lower priority requests. $M/G/1$ queues with EDF scheduling achieved better waiting times for lower priority requests, compared to simple $M/G/1$ queues.

Comparing EDF based scheduling to other popular scheduling techniques implemented in a $M/G/1$ system reveals that EDF, while achieving such service differentiation still manages to record an acceptable deadline loss rate. Non-differentiating algorithms such as FCFS and round-robin scheduling achieve better waiting times overall, as they do not favour a particular class of requests. However, this results in a higher deadline loss rate. Moreover, compared to an algorithm with a static priority ordering, EDF being a dynamic priority model achieved a better balance between shorter waiting times for higher priority requests and waiting times with no indication of over-starvation, for lower priority requests. Moreover, the evaluation confirmed our claim that explicit scheduling of requests based on a deadline will give the system a chance of achieving better execution time predictability and QoS.

While $M/G/1/./EDF$ model was presented within the context of web services, it can be used for any system or application that uses EDF scheduling as deadlines could be used in applications for any time related activity, not limiting to execution. The result of the evaluation presented, will hold true for any such system.

Overall, the evaluations confirmed that the research goals of achieving predictability of execution in stand-alone and cluster based web service deployments, were fulfilled. Moreover, they confirm that the development platforms and operating systems we chose to support our solutions indeed provide the proper features required for predictability. The intended behaviour of differentiated request processing could be observed in the empirical results. While there are task rejections that happen especially in high traffic conditions, empirical results show that they could be decreased by adding more server resources. Due to operational constraints, the conducted evaluations were limited up to just 4 server machines that had desktop grade hardware. However, with the empirical data it can be concluded that adding more resources would guarantee the reduction of task rejections while maintaining the high deadline achievement rates by the solutions presented.

Within the related-work discussed for both stand-alone and cluster based web services, a commonality is the consideration of execution time as a QoS parameter. Various approaches are taken to achieve some level of differentiation in request processing in all

of them and many try to achieve a probability based measure of execution time among different classes of requests. Many of the try to dynamically adjust the ratio of request processing to meet the pre-defined levels of processing outlined in an SLA. While, these techniques may be successful in meeting the overall perceived levels of performance when requests being processed over a period of time is considered, none of them can guarantee the same execution times in a consistent manner for every service invocation. Therefore, by design all of them fail to achieve predictable execution times in a repeatable and a consistent manner. Such levels of predictability can only be achieved if requests are purposely scheduled to ensure a deadline in a definitive manner. Moreover, the middleware must be designed ground-up with the support required to achieve this level of predictability, from the libraries, development platform and the operating system being used. Additionally, the acceptance of a request for execution must be validated for schedulability ensuring both its deadline and that of the others executing within the same server.

Req.	Sharma et al. [1] Tien et al. [2]	Pacifici et al. [3]	Gmach et al. [4]	Cao J. et al. [5]	García et al. [6]	Helander et al. [7] Mathes et al. [8]	Our Method
G1	⊗	⊗	⊗	⊗	⊗	⊗	●
G2	⊗	⊗	◐	⊗	◐	●	●
G3	⊗	◐	⊗	⊗	◐	◐	●
G4	●	●	●	●	●	◐	●
G5	⊗	⊗	⊗	⊗	⊗	◐	●

⊗ Not Compliant, ◐ Partially, ● Fully

[1] [Sharma et al., 2003] [2] [Ching-Ming Tien, 2005] [3] [Pacifici et al., 2005] [4] [Gmach et al., 2008]
[5] [Cao et al., 2010] [6] [García et al., 2009] [7] [Helander and Sigurdsson, 2005] [8] [Mathes et al., 2009a]

Table 7.1: Compliance of related work to predictability requirements

With Table 7.1, some of the related work on web services are validated against the predictability requirements presented in Chapter 5. As discussed previously, many of them satisfy guideline G4, having some method of service differentiation. Some of them making conscious selection of requests for execution based on statistics, partially meet with guideline G3. Conscious scheduling of requests based on a perceived end result partially meets with guideline G2. However, none of them are compliant with guideline G1 nor fully compliant with all guidelines identified.

7.3 Future Work

This section outlines the potential paths for future work that stems from our research.

Predictability in the network layer

The proposed approach for achieving predictability of execution in stand-alone middleware included the deadline based scheduling of requests and the laxity based schedulability check. We made the assumption that web service requests experience no delays on the network. However, data travelling within an active network is bound to experience some delay. Depending on the type of network web services are used on, the delay experienced by requests maybe quite significant (for instance on the Internet). Moreover, the time spent on the network transmission maybe significant compared to the execution time of a service.

Given these circumstances, an area of future work would be to achieve differentiated request transmission and in turn predictability on the network. Current network infrastructure and protocols may have features that support this purpose. Specialised network architectures such as IntServ [Braden et al., 1994; TACS, 2012a] and DiffServ [Blake et al., 1998; Cisco Systems, 2005; TACS, 2012b] that enable QoS based bandwidth reservation, are already in place. Moreover, the possibility of using protocols such as Resource Reservation Protocol (RSVP) [Metz, 1999; White, 1997] and Multi-Protocol Label Switching (MPLS) [Awduche, 1999; Davie and Rekhter, 2000] can be investigated further on. One possible way of moving towards network level predictability is to use routers that support such protocols and implement a priority structure on top of them. The middleware and the client components could be given the ability to negotiate the deadlines with the router and receive prioritised data transfer. This may require web services middleware to reside on routing nodes in order to negotiate priorities at the application layer of the network protocol stack.

Extending predictability of execution across application boundaries

One of the assumptions made in Chapter 1 is the focus of this research being only on the execution of service invocations within the boundary of the web services middleware. As mentioned, service invocations may result in executions going beyond application

boundaries. A database query, the use of business objects through an application server and composite services are some examples for such execution.

Achieving predictability of execution in these software layers are potential areas for future work. The architecture of each application will be unique and there may not be a generic solution that is applicable to all. However, the guidelines we provided in chapter 5 can be used to identify the changes required and as a check-list thereafter, in the enhancement process. Achieving predictability in these applications would require them to use a priority structure. A service invocation that spans across multiple application boundaries will have to be priority negotiated and the different applications must coordinate to have the processing resources available in order to meet the overall deadline requirement. Given the different architectures in each of them, achieving predictability in each type of application will be research areas on their own.

Reducing request rejections through selective re-transmission

The laxity based schedulability check we use for the admission control, rejects requests based on potential deadline misses. In the scope of this research, they were simply rejected and reported back to the client applications. It may be worth investigating the possibility of re-transmission of some of these requests that may have a chance if re-checked for schedulability, with the same or a different executor after a period of time. Within this period, there is a chance that processing resources may free-up at the server and therefore the target requests maybe accommodated.

Dispatching algorithms in the likes of RT-Sequential is a potential area of research. Identifying potential requests for retransmission (without considering every rejected request) and supporting multiple servers are useful aspects to research further on.

Custom-built web services middleware

The implementations we presented in chapter 5 were enhancements made to existing middleware products widely in use. Given our goal of achieving predictability of execution or support for it in all levels of software in a bottom-up manner, a custom built web services middleware will achieve the best level of performance.

While we achieved acceptable levels of performance with our implementations, they contain code that maybe sub-optimal given request processing and execution. An in-

teresting extension to our research would be building web services middleware from scratch with purposefully designed code in all layers and components of the application. Given the limited time period for this research and the effort involved, it was difficult for us to achieve this in the time period available. However, such a middleware is bound to have better performance levels than our enhanced versions and will result in better design patterns and software engineering techniques that could become useful in achieving predictability in other types of applications.

Improved performance model for preemptive $M/G/1/./EDF$

We identified deadline misses that took place in the simulation runs to be the main contributing factor for the difference between analytical and simulation results. This phenomenon is not accounted for by any of the variables used in our performance model. An interesting research area would be to have a representation of a deadline miss in our performance model.

In the related work section of chapter 6, we discuss a few attempts by other researchers in quantifying the loss rate for a $M/G/1$ type queues. Although their models are only focused on loss rate minimisation, it may be possible to build on their work and define the loss rate and quantify its effect on the waiting time of requests. This is another interesting area for future work.

Performance models for preemptive $G/G/1$

The analytical model we presented in chapter 6 allowed a general service time distribution. However, we made the assumption that request arrivals were following a Poisson process. As a result, we were able to build on existing definitions [Kleinrock, 1975, 1976] for the type of system we envisioned.

While, Poisson arrivals are a better representation for bursty requests, an interesting research avenue would be to define a similar performance model considering request arrivals to be general. Such a performance model can be used to approximate the performance of EDF in various different environments, not being limited to web services or Internet traffic.

Bibliography

Apache Software Foundation. Apache Synapse. http://synapse.apache.org/, 9 June 2008. Last Accessed - 13/07/2012. 7, 100, 137

Apache Software Foundation. Apache Axis2. http://ws.apache.org/axis2/, 8 June 2009. Last Accessed - 13/07/2012. 6, 7, 37, 61, 136

K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*. Addison-Wesley Professional, 2006. 6, 132

G. Arora. Automated Analysis and Prediction of Timing Parameters in Embedded Real-Time Systems Using Measured Data. Master's thesis, Electrical Engineering Dept., University of Maryland, College Park, June 1997. 29

D. Awduche. MPLS and Traffic Engineering in IP Networks. *Communications Magazine, IEEE*, 37(12):42–47, 1999. 220

A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle. Multi-tenant SOA middleware for cloud computing. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 458–465. IEEE, 2010. 5, 36

C.-P. Bezemer and A. Zaidman. Multi-tenant SaaS applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 88–92, New York, NY, USA, 2010. ACM. 5

S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and Weiss W. An Architecture for

Differentiated Services. http://tools.ietf.org/html/rfc2475, Dec. 1998. Last Accessed - 03/06/2012. 220

G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley-Blackwell, 2006. 180, 190

D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, M. Hadley, C. Kaler, D. Langworthy, F. Leymann, B. Lovering, S. Lucco, S. Millet, N. Mukhi, M. Nottingham, D. Orchard, J. Shewchuk, E. Sindambiwe, T. Storey, S. Weerawarana, and S. Winkler. Web Services Addressing (WS-Addressing). http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/, 10 Aug. 2004. Last Accessed - 18/05/2012. 24

R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: An Overview. http://tools.ietf.org/html/rfc1633, June 1994. Last Accessed - 03/06/2012. 220

E. Bruno and G. Bollella. *Real-time Java programming: with Java RTS*. Prentice Hall PTR, 2009. 127

G. C. Buttazzo. *Hard real-time computing systems : Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997. 28, 29, 30, 37

J. Cao, H. Zhao, M. Li, and J. Wang. A dynamically self-configurable service process engine. *World Wide Web*, 13(4):475–495, 2010. 12, 81, 219

V. Cardellini, M. Colajanni, and P. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999. 78

V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The state of the art in locally distributed Web-server systems. *ACM Computing Surveys (CSUR)*, 34(2):263–311, 2002. 77, 78

V. Cardellini, M. Colajanni, and P. Yu. Request redirection algorithms for distributed web systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(4): 355–368, 2003. 11, 80

J. Carlstrom and R. Rom. Application-aware admission control and scheduling in web servers. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE*

Computer and Communications Societies. Proceedings. IEEE, volume 2, pages 506–515. IEEE, 2002. 11, 41

E. Casalicchio, V. Cardellini, and M. Colajanni. Content-Aware Dispatching Algorithms for Cluster-Based Web Servers. *Cluster Computing*, 5(1):65–74, 2002. 80

E. Cerami and S. St Laurent. *Web services essentials*. O'Reilly & Associates, Inc., 2002. 4, 18

D. Chapell. Windows Communications Foundation. http://msdn.microsoft.com/library/ee958158.aspx, Mar. 2010. Last Accessed - 06/07/2012. 6, 37

D. Chappell and T. Jewell. Java web services: using Java in service-oriented architectures, 2002. 18, 19

K. Chen and L. Decreusefond. An approximate analysis of waiting time in multi-class m/g/1/./edf queues. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 190–199, New York, NY, USA, 1996. ACM. 13, 15, 165, 166, 169, 171, 172, 180, 181, 192, 193, 195

P. Ching-Ming Tien, Cho-Jun Lee. SOAP Request Scheduling for Differentiated Quality of Service. In *Web Information Systems Engineering - WISE Workshops*, pages 63–72. Springer Berlin / Heidelberg, Oct. 2005. 11, 40, 219

E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315, 15 Mar. 2001. Last Accessed - 13/05/2012. 18

G. Ciardo, A. Riska, and E. Smirni. EquiLoad: a load balancing policy for clustered web servers. *Performance Evaluation*, 46(2-3):101–124, 2001. 11, 81

Cisco Systems. DiffServ – The Scalable End-to-End QoS Model. http://www.cisco.com/en/US/technologies/tk543/tk766/technologies_white_paper09186a00800a3e2f.html, Aug. 2005. Last Accessed - 03/06/2012. 220

D. Clark. Fiber-based metropolitan access networks for internet traffic. In *Optical Fiber Communication Conference, 2000*, volume 2, pages 44–46. IEEE, 2000. 165

E. Coffman Jr, R. Muntz, and H. Trotter. Waiting time distributions for processor-sharing systems. *Journal of the ACM (JACM)*, 17(1):123–130, 1970. 125, 127

M. Colajanni and P. Yu. A performance study of robust load sharing strategies fordistributed heterogeneous Web server systems. *Knowledge and Data Engineering, IEEE Transactions on*, 14(2):398–414, 2002. 11, 80

T. Dag and O. Gokgol. A priority based packet scheduler with deadline considerations. In *Communication Networks and Services Research Conference, 2006. CNSR 2006. Proceedings of the 4th Annual*, pages 8–pp. IEEE, 2006. 168

B. Davie and Y. Rekhter. MPLS: technology and applications. 2000. 220

D. Davis, A. Karmarkar, G. Pilz, S. Winkler, and U. Yalcinalp. OASIS Web Services Reliable Messaging Protocol. http://docs.oasis-open.org/ws-rx/wsrm/200608/wsrm-1.1-spec-cd-04.pdf, 7 Aug. 2006. Last Accessed - 18/05/2012. 24

M. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 74:807–813, 1974. 32

P. Dibble. *Real-time Java platform programming*. Prentice Hall PTR, 2002. 127

D. Dyachuk and R. Deters. Transparent scheduling of web services. In *3rd International Conference on Web Information Systems and Technologies*, 2007. 11, 41

L. Eggert and J. Heidemann. Application-level differentiated services for Web servers. *World Wide Web*, 2(3):133–142, 1999. 82

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*, pages 276–286. ACM, 2004. 11, 41

T. Erl. *SOA: Principles of Service Design*. Prentice Hall Press, 2007. 20

A. Erradi and P. Maheshwari. wsBus: QoS-aware middleware for reliable web services interactions. *e-Technology, e-Commerce and e-Service, 2005. EEE '05. Proceedings. The 2005 IEEE International Conference on*, pages 634–639, March-1 April 2005. 12, 41, 129

R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000. 19

V. Gamini Abhaya, Z. Tari, and P. Bertok. Achieving Predictability and Service Differentiation in Web Services. In *ICSOC-ServiceWave '09: Proceedings of the 7th International Conference on Service-Oriented Computing*, pages 364–372. Springer, 2009. 35

V. Gamini Abhaya, Z. Tari, and P. Bertok. Using Real-Time Scheduling Principles in Web Service Clusters to Achieve Predictability of Service Execution. In *Service-Oriented Computing: 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7-10, 2010. Proceedings*, pages 197–212. Springer, 2010a. ISBN 3642173578. 76

V. Gamini Abhaya, Z. Tari, and P. Bertok. Building web services middleware with predictable service execution. In *Web information systems engineering-WISE 2010: 11th international conference, hong kong, china, december 12-14, 2010, proceedings*, volume 6488, page 23. Springer-Verlag New York Inc, 2010b. 35, 76, 124

V. Gamini Abhaya, Z. Tari, and P. Bertok. Building web services middleware with predictable execution times. *World Wide Web*, pages 1–60, 2012. 35, 76, 124

V. Gamini Abhaya, Z. Tari, P. Bertok, and P. Zeephongsekul. Waiting Time Analysis for a Multi-class Preemptive M/G/1/./EDF queue. *Journal of Parallel and Distributed Computing*, 2013. (In Submission). 163

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. 145, 156

D. F. García, J. García, J. Entrialgo, M. García, P. Valledor, R. García, and A. M. Campos. A qos control mechanism to provide service differentiation and overload protection to internet scalable servers. *IEEE Transactions on Services Computing*, 2

(1):3–16, 2009. ISSN 1939-1374. doi: http://doi.ieeecomputersociety.org/10.1109/TSC.2009.3. 12, 78, 81, 219

Gartner and Forrester. Use of Web services skyrocketing. http://utilitycomputing.com/news/404.asp, 30 Sept. 2003. 4

K. Gilly, C. Juiz, and R. Puigjaner. An up-to-date survey in web load balancing. *World Wide Web*, pages 1–27, 2011. 78

D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper. Adaptive quality of service management for enterprise services. *ACM Transactions on the Web (TWEB)*, 2(1):1–46, 2008. 12, 81, 219

S. Graham, D. Davis, S. Simeonov, Daniels G., Brittenham P., Y. Nakamura, Fremantle P., Konig D., and Zentner C. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI*. Sams Publishing, Indianapolis IN USA, 2nd edition, 8 July 2004. 6, 19, 37, 126

M. Gudgin, N. Mendelsohn, M. Nottingham, and H. Ruellan. SOAP Message Transmission Optimization Mechanism. http://www.w3.org/TR/2005/REC-soap12-mtom-20050125/, 25 Jan. 2005. Last Accessed - 14/05/2012. 19

M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). http://www.w3.org/TR/2007/REC-soap12-part1-20070427/, 27 May 2007a. Last Accessed - 17/05/2012. 23

M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. SOAP Version 1.2 Part 2: Adjuncts (Second Edition). http://www.w3.org/TR/2007/REC-soap12-part2-20070427/, 27 May 2007b. Last Accessed - 17/05/2012. 23

A. Gurugé. *Web services: theory and practice*. Digital Pr, 2004. 18

M. Harchol-Balter, M. Crovella, and C. Murta. On Choosing a Task Assignment Policy for a Distributed Server System. *Journal of Parallel and Distributed Computing*, 59(2):204–228, 1999. 11, 81

P. Heidelberger. Fast simulation of rare events in queueing and reliability models. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 5(1):43–85, 1995. 186

J. Helander and S. Sigurdsson. Self-tuning planned actions time to make real-time SOAP real. *Object-Oriented Real-Time Distributed Computing, ISORC. Eighth IEEE International Symposium on*, pages 80–89, 2005. 8, 12, 41, 42, 129, 130, 219

J. Jackson. *Scheduling a Production Line to Minimize Maximum Tardiness*. University of California, 1955. 32

D. Jayasinghe and A. Azeez. *Apache Axis2 Web Services*. Packt Pub Limited, 2011. 19

M. Kargahi and A. Movaghar. A method for performance analysis of earliest-deadline-first scheduling policy. *The Journal of Supercomputing*, 37(2): 197–222, 2006. 12, 165, 168

N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web services choreography description language version 1.0. http://www.w3.org/TR/ws-cdl-10/, 9 Nov. 2005. Last Accessed - 12/07/2012. 22

R. Khalaf and W. Nagy. Business process with BPEL4WS: Learning BPEL4WS, part 2. http://www.ibm.com/developerworks/webservices/library/ws-bpelcol2/, 1 Aug. 2002. Last Accessed - 12/07/2012. 22

M. Kihl, A. Robertsson, M. Andersson, and B. Wittenmark. Control-theoretic analysis of admission control mechanisms for web server systems. *World Wide Web*, 11(1): 93–116, 2008. 82

L. Kleinrock. Queueing Systems. Volume 1: Theory. 1975. 222

L. Kleinrock. Queueing Systems Volume 2: Computer Applications, 1976. 167, 169, 173, 176, 179, 180, 190, 222

E. Kresch and S. Kulkarni. A poisson based bursty model of internet traffic. In *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, pages 255 –260, 31 2011-sept. 2 2011. 165

J. Lehoczky. Real-time queueing theory. In *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 186–195. IEEE, 1996. 13, 168

J. Lehoczky. Using real-time queueing theory to control lateness in real-time systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 25, pages 158–168. ACM, 1997. 169

W. Li, K. Kavi, and R. Akl. A non-preemptive scheduling algorithm for soft real-time systems. *Computers & Electrical Engineering*, 33(1):12–29, 2007. 12, 168

Q. Liang, X. Wu, and H. Lau. Optimizing service systems based on application-level qos. *Services Computing, IEEE Transactions on*, 2(2):108–121, 2009. 40

C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321738.321743. 30

E. Marks and M. Werrell. *Executive's guide to web services*. John Wiley & Sons, Inc., 2003. 18

M. Mathes, J. Gartner, H. Dohmann, and B. Freisleben. Soap4ipc: A real-time soap engine for industrial automation. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 220–226, Feb. 2009a. doi: 10.1109/PDP.2009.21. 8, 12, 41, 42, 130, 219

M. Mathes, C. Stoidner, S. Heinzl, and B. Freisleben. Soap4plc: web services for programmable logic controllers. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 210–219. IEEE, 2009b. 130

M. Mathes, C. Stoidner, R. Schwarzkopf, S. Heinzl, T. Dörnemann, H. Dohmann, and B. Freisleben. Time-constrained services: a framework for using real-time web services in industrial automation. *Service Oriented Computing and Applications*, 3 (4):239–262, 2009c. 8, 12, 130

C. Metz. Rsvp: general-purpose signaling for ip. *Internet Computing, IEEE*, 3(3): 95–99, 1999. 220

Microsoft Corporation. [MS-DCOM]: Distributed Component Object Model (DCOM) Remote Protocol Specification. http://msdn.microsoft.com/library/cc201989.aspx, 28 Mar. 2012. Last Accessed - 17/05/2012. 4, 22

N. Mitra and Y. Lafon. SOAP Version 1.2 Part 0: Primer (Second Edition). http://www.w3.org/TR/2007/REC-soap12-part0-20070427/, 27 Apr. 2007. Last Accessed - 17/05/2012. 23

Mohammadi A. and Akl S. G. Scheduling Algorithms for Real-Time Systems. Technical report, School of Computing, Queen's University, Kingston, Ontario, Canada, 15 July 2005. 29, 31

Mor Harchol-Balter. Task Assignment with Unknown Duration. *Journal of the ACM*, 49(2):260–288, Mar. 2002. 80

S. Natarajan and W. Zhao. Issues in building dynamic real-time systems. *IEEE Software*, 9(5):16–21, 1992. 37

OASIS. OASIS Web Services Security (WSS) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss, 28 Nov. 2006. Last Accessed - 18/05/2012. 24

Oracle. Java Remote Method Invocation (Java RMI). http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/index.html, 2010. Last Accessed - 17/05/2012. 4, 22

Oracle Corporation. Sun Java Real-time System. http://java.sun.com/javase/technologies/realtime/, 2009a. 127, 132, 137, 147

Oracle Corporation. Thread Scheduling Visualizer 2.0 - Sun Java RealTime Systems 2.2. http://java.sun.com/javase/technologies/realtime/reference/TSV/JavaRTS-TSV.html, 21 Aug. 2009b. 160

G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster-based web services. *Selected Areas in Communications, IEEE Journal on*, 23(12):2333–2343, Dec. 2005. ISSN 0733-8716. doi: 10.1109/JSAC.2005.857208. 12, 78, 81, 219

M. Papazoglou. *Web services: principles and technology*. Addison-Wesley, 2008. 18, 19, 20, 22

C. Peltz. Web services orchestration and choreography. *IEEE - Computer*, 36(10): 46–52, 2003. 22

I. Pyarali, D. Schmidt, and R. Cytron. Techniques for enhancing real-time CORBA quality of service. *Proceedings of the IEEE*, 91(7):1070–1085, 2003. 126

S. Ran. A model for web services discovery with QoS. *ACM SIGecom Exchanges*, 4 (1):1–10, 2003. 11, 39

RedHat Inc. JBoss Application Server. http://www.jboss.com/products/platforms/application/, 2009. 7

L. Richardson and S. Ruby. *RESTful web services*. O'Reilly Media, 2007. 19

D. Schmidt and F. Kuhns. An overview of the Real-Time CORBA specification. *Computer*, 33(6):56–63, 2000. 126

D. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar. A high-performance end system architecture for real-time CORBA. *IEEE Communications Magazine*, 35(2): 72–77, 1997. 126

K. Scribner, K. Scribner, and M. C. Stiver. *Understanding Soap: Simple Object Access Protocol*. Sams, Indianapolis, IN, USA, 2000. ISBN 0672319225. 22

A. Sharma, H. Adarkar, and S. Sengupta. Managing QoS through prioritization in web services. *Web Information Systems Engineering Workshops, Proceedings.*, pages 140–148, Dec. 2003. doi: 10.1109/WISEW.2003.1286796. 11, 40, 219

M. Spuri. Earliest Deadline scheduling in real-time systems. *Doctorate Dissertation, Scuola Superiore S. Anna, Pisa*, 1995. 34

J. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Systems*, 28 (2):237–253, 2004. ISSN 0922-6443. 30, 132

J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.7053. 37

J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo. *Deadline scheduling for real-time systems : EDF and related algorithms*. Kluwer Academic Publishers, 1998. 28, 29, 30, 33, 37, 135

V. Stantchev. Performance evaluation of cloud computing offerings. In *Advanced Engineering Computing and Applications in Sciences, 2009. ADVCOMP '09. Third International Conference on*, pages 187 –192, oct. 2009. 125, 127

Subramaniam V. *Programming Concurrency on the JVM - Mastering Synchronization, STM, and Actors*. Pragmatic Bookshelf, Aug. 2011. 119, 125, 127, 147

Sun Microsystems. Glassfish Application Server - Features. http://www.oracle.com/us/products/middleware/application-server/oracle-glassfish-server/index.html, 2009. 6, 7, 37

TACS. Int-Serv Architecture. http://transanatolia.eu/Analyses/Internet/ip%20qos%20architectures/int-serv_architecture.htm, 27 May 2012a. Last Accessed - 03/06/2012. 220

TACS. DiffServ Architecture. http://transanatolia.eu/Analyses/Internet/ip%20qos%20architectures/diff-serv_architecture.htm, 27 May 2012b. Last Accessed - 03/06/2012. 220

M. Tian, A. Gramm, T. Naumowicz, H. Ritter, and J. Freie. A concept for QoS integration in Web services. *Web Information Systems Engineering Workshops, Proceedings.*, pages 149–155, 2003. 11, 40

W.-T. Tsai, X. Sun, and J. Balasooriya. Service-oriented cloud computing architecture. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 684 –689, april 2010. doi: 10.1109/ITNG.2010.214. 5

A. Varga. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, volume 9, 2001. 185

A. Varga. Omnet++. *Modeling and Tools for Network Simulation*, pages 35–59, 2010. 185

S. Vinoski. Distributed Object Computing with CORBA. *C++ Report*, 5(6):32–38, 1993. 4, 22

W. Vogels. Web services are not distributed objects. *Internet Computing, IEEE*, 7(6): 59–66, 2003. 19

A. J. Wang and V. Baglodi. Evaluation of java virtual machines for real-time applications. *Journal of Computing Sciences in Small Colleges*, 17(4):164–178, 2002. ISSN 1937-4771. 137

N. Wang, D. Schmidt, and D. Levine. Optimizing the CORBA Component Model for High-performance and Real-time Applications. *Work-in-Progress session at the Middleware 2000 Conference*, 2000. 126

S. Weerawarana and F. Curbera. Business process with BPEL4WS: Understanding BPEL4WS, part 1. http://www.ibm.com/developerworks/webservices/library/ws-bpelcol1/, 1 Aug. 2002. Last Accessed - 12/07/2012. 22

S. Weerawarana, R. Chinnici, M. Gudgin, and J. Moreau. Web Services Description Language (WSDL) Version 1.2. http://www.w3.org/TR/2002/WD-wsdl12-20020709, 9 July 2002. Last Accessed - 13/05/2012. 18

S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR, 2005. 18, 19

P. White. Rsvp and integrated services in the internet: A tutorial. *Communications Magazine, IEEE*, 35(5):100–106, 1997. 220

L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Sheng. Quality driven web services composition. *Proceedings of the 12th international conference on World Wide Web*, pages 411–421, 2003. 11, 40

L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, H. Chang, I. Center, and N. Yorktown Heights. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004. 11, 40