Enhancing SOAP Performance by achieving Service Differentiation and Predictability

Vidura Gamini Abhaya
RMIT University
Melbourne, Victoria, Australia
E-mail: vabhaya@cs.rmit.edu.au

Abstract

Simple Object Access Protocol (SOAP) based web services have become a widely used middleware for heterogeneous distributed systems over the years. Although the driving factor of its popularity has been the simple text based nature of eXtensible Markup Language (XML) SOAP, the same reason has resulted web services being overlooked for certain classes of applications. To this date, binary based middleware such as Common Request Broker Architecture (CORBA) is still preferred for applications that require predictability in remote invocations. This document presents an approach for achieving predictability in SOAP based web services by means of service differentiation. This research proposes a model and a priority based Real-time scheduling algorithm based on Real-time scheduling principles. The proposed algorithm schedules requests based on a deadline which would be associated with each request received by the server. Every request is subjected to an acceptance test which checks for schedulability of the request. The request is only accepted for execution if the associated deadline requirement could be met and only if the request does not compromise the deadlines of the tasks already accepted and active in the system. Upon not being able to satisfy these conditions, the request will be rejected. The accepted requests are prioritized based on their deadline requirements, giving priority to the earliest deadline. The proposed model and algorithm is empirically evaluated by implementing it on the widely used SOAP engine Apache Axis2. To ensure predictability a Real-time Operating System in the form of Sun Solaris and a development platform supporting Real-time development in the form of Sun Real-time Java is used in the implementation. The implementation is benchmarked against the unmodified version of Axis2 for various types of workloads and arrival rates, given different deadlines. The workloads are characterized by probability distributions and the number of tasks accepted, the number of deadlines achieved and their execution times are considered as metrics of performance. This document provides a brief overview of principles behind real-time scheduling, and a detailed discussion on the proposed model and algorithm followed by the implementation details on Axis2. Finally challenges faced in implementation and testing it presented followed by the results of the performance comparison. The results confirm that the proposed technique can be used to benefit applications requiring predictability in execution.

1 Introduction

Web Services are software components with discrete functionality that expose business logic to other applications, on a network. Consumers of a web service uses the web service through
the Application Programming Interface (API) provided, using network protocols to transfer
the data between the two applications. Therefore it is essentially a middleware that bridges
the communication between two or more applications. Due to its textual nature and the use
of Hyper Text Transfer Protocol (HTTP) for transport, it has become the de-facto method
for integrating geographically distributed information systems [1].

Simple Object Access Protocol (SOAP), the web service messaging protocol is based on
Extensible Markup Language (XML) and HTTP. Due to its verbose nature, the ratio between
the size of actual data in a SOAP packet and non-data elements, are seen as a drawback on
it’s usage [2]. Hence, binary based middleware such as CORBA [3] are still being used in
distributed systems. For applications with real-time requirements and predictability in execution
times, binary based middleware has been the default choice. Specialised implementations
such as Real-Time CORBA [4] takes a step further into ensuring these requirements are met
to its entirety.

A Web Service, like all Remote Procedure Call (RPC) mechanisms have a step of mar-
shalling parameters at both client and server ends. These are referred to as encoding/decoding
and serialization/deserialization. In order to maintain transparency to the client and the
server, these are usually handled by Web Service libraries or by a SOAP engine. Attempts
of enhancing the performance of these steps can be seen in the work of Abu-Ghazaleh N.
et. al [5, 6], Suzumara T. et. al. [7] and Werner C. et. al. [2], by using techniques such
as differential deserialization, differential encoding, compression and pipelining techniques at
HTTP level. A different approach to this is to introduce Quality of Service (QoS) parameters
to Web Service Architecture. Currently available implementations of SOAP and Web Service
libraries such as JAX-WS, Microsoft .NET and SOAP engines such as Apache Axis2, gSOAP
has a best effort approach to servicing requests where, as many requests as possible are exe-
cuted in parallel to increase throughput. Although the client applications might have certain
a deadline or performance requirements, meeting them would not be given a higher priority.
Research into incorporating QoS requirements to this process has been carried out by a few.

To ensure QoS requirements are met, many researchers have introduced QoS brokers,
proxies and middleware between the Web Service and the client. Tian M., Gramm A. et
al. [8] and Ran S. [9] in their work, introduces a QoS service broker that a client queries
with QoS parameters before the actual Web Service call is made. Zeng L., Benatallah B. et
al. [10] proposes a Web Service Quality model where the QoS parameters are ensured when
composition of a execution plan is made. While these solutions brings in the aspect of QoS
into the consideration, they have no means of guaranteeing these at the time of execution.
Approaches such as in the work of Tien C.M., Lee C.J, et al. [11], Sharma A., Adarkar A.
and Sengupta S. [12] and Helander J., Sigurdsson S. [13] try to achieve differentiation at the
execution level. These will be discussed further under Related Work.

For applications with real-time requirements, predictability of execution is the most im-
portant factor. Although some of the attempts mentioned above tries to achieve certain
QoS parameters, they fail to guarantee predictability at the execution level. The work that
comes close to achieving it does it in a confined embedded environment where tasks and
their resource requests are pre-defined and remain static. The challenge would be to achieve
predictability in the totally dynamic environment that web services are used in.

The proposed solution consists of a model and an algorithm developed by extending
principles in real-time scheduling algorithms. The solution follows a policy of only accepting
requests that could be guaranteed to meet a deadline that it is associated with. The solution
proposes an acceptance test that checks the schedulability of all incoming requests. This is
carried out at runtime against each incoming request, to ensure that by accepting the request not only the deadline associated with it could be met, but also it would not compromise the deadlines of the requests already accepted and active in the system. Moreover, a real-time scheduling algorithm is introduced to ensure that the requests are scheduled in an appropriate manner, leading to satisfying their deadline requirements. The proposed model and algorithm is implemented in a real-life, widely used SOAP engine Apache Axis2. The implementation is thoroughly tested for various types of workloads and arrival times to ensure all objectives are met.

The rest of the document is structured as follows. The next section discusses the related work in detail. In Section 3, Real-time scheduling, real-time scheduling algorithms and the principles behind them are introduced. Section 4, discusses the proposed solution in detail and a sample scenario further illustrates the application of the proposed model and algorithm. In Section 5 the architecture of Apache Axis2 and the implementation of the proposed model and algorithm in Axis2 is discussed in detail. Section 6 presents the challenges faced in evaluating the implementation and details how it was tested. Section 7 presents the results of the various experiment runs followed by a detailed discussion of the results. In Section 8 a summary of the entire research and a conclusion is provided. Throughout this document, the words request and task is used interchangeably and refers to a web service request by a client to a target web service.

2 Related Work

This section briefly describes attempts by other researchers to Integrate QoS properties to Web Services and higher level service compositions.

2.1 Service Differentiation based on QoS

If a service provider needs to differentiate the levels of service they offer to consumers, the requests received from the consumers need to be treated differently. In order to differentiate how requests are serviced, a mechanism is needed to separate out requests based on some criteria. This criteria could be certain QoS parameters a request is associated with. Several needs arise in fulfilling these requirements. Clients need a way to convey their QoS requirements. Service providers need a way to describe levels of QoS that could be promised by each service hosted. Hence, a well defined and accepted way of describing the QoS properties is needed. Furthermore, consumers require ways of finding services that meets their QoS needs and most importantly, ways of meeting desired QoS levels in processing and execution needs to be in place at the service providers. The following highlight recent work in these areas by various researchers.

Many attempts on describing the QoS requirements of a client and what could be expected from a service provider, can be seen in the recent past. These include Web Service Level Agreement Standard (WSLA) by IBM [14][15], Web Services Offer Language (WSOL) [16] and solutions based on WS-Policy[17]. Using them, makes it possible to define QoS metrics in a standard way that is widely accepted and understood. Work by Ran S. [9] attempts to facilitate the discovery of services, upon querying by a client with QoS requirements as the criteria. A modified Universal Description, Discovery and Integration (UDDI) registry introduced, stores QoS levels offered by services along with their functional descriptions. A module that acts as a ‘broker’ is introduced between the client and the UDDI registry. The
module verifies the claims of QoS levels promised by services and certifies it to clients. Hence, there’s is an additional step involved in the lookup process. The frequency and the method of verification has not been clearly discussed. While this would not be an issue when the QoS properties are non-functional such as related to Security, Price etc., depending on how the functional properties such as execution time is ensured, it can be of a concern.

A similar approach has been taken by Tian M., Gramm A. et al. [8], in introducing a QoS Broker between the client and the UDDI registry. The main difference with the work by Ran S. being the client interaction between the client and the UDDI been replaced by the interaction with the QoS broker. Furthermore, their solution also introduces an ontology in describing the QoS properties. Moreover, their model also proposes web service proxies that map the requests and QoS properties to the underlying network layer. the proxies mark the priority levels of the requests in the IP packets and allow the network to control QoS depending on them. While this attempts to maintain the priority levels all the way to the wire, the solution relies on the ability of the network devices to support such QoS properties. However, how to ensure functional QoS is met at the service provider’s has not been widely discussed. Another broker based approach is employed by Yu T. and Lin K.J. [18] where the QoS broker is implemented as part of the service provider. The brokering module maintains QoS information about the services hosted, with periodic updates. It has a submodule that negotiates with clients on the level of QoS the services could provide, and an Admission control submodule which enforces the granted QoS levels to the client, on the server. Furthermore, the solution also involves a submodule to analyze the statistics after a task has been finished and use the data has feedback by the negotiation submodule. The server resource allocation is done with the use of 2 algorithms. A Homogeneous resource allocation algorithm (HQ) tries to evenly divide the available resources among all clients using a particular service, assuming that all clients have the same minimum QoS requirement. A Non-homogeneous resource allocation algorithm (RQ) is also employed which allocates different amounts of resources to different clients according to their QoS requirements. A virtual client is created and used to allocate unused resources to ensure incoming requests can be catered for. However, in the event of the reserved resources not being enough for an incoming client, the resource allocation is reconfigured among the existing clients to satisfy the incoming client. Although the algorithm tries to achieve the highest system utilization with the minimum reconfigurations possible, the reconfiguration step becomes a huge overhead. The approach by Zeng L., Benatallah B. et al. [10] takes it to a higher level where multiple services are composed depending on the QoS requirements. Each service promises certain QoS levels that would be met. In choosing between web services to carry out a workflow, these QoS levels are used to pick the best one for the case, depending on a couple of algorithms used. These approaches introduces the notion of QoS, yet fails to ensure that the required levels would be met at execution level.

Sharma A., Adarkar A. and Sengupta S. [12] takes a different approach of differentiation by means of regulating the throughput of the types of requests. Requests are classified into service classes based on conditions such as the nature of application (i.e. a stock trading service versus a price inquiry), the device being used (i.e. a Personal Computer versus a mobile device), nature of client (i.e. a paying customer versus a free request), etc. Based on these conditions, priorities are assigned to each of the categories. Depending on the current arrival rate of these categories, the priorities are dynamically adjusted with the use of a penalty function, to reduce starvation. A lower than normal arrival rate penalizes the priority whereas a rate higher than the expected level with enforce it positively. This solution achieves some level of differentiation in the throughput of the requests. However, the guarantee of completing
the execution of a service within a certain time period cannot be predicted. Moreover, the request classification process can be an overhead, which has not been measured.

Tien C.M., Lee C.J., et al. [11] follows a similar approach in classifying requests into QoS classes based on Service Level Agreements (SLA) between service providers and clients. Processing requirements for each function is profiled offline and the data is used to calculate the actual processing requirement of each request. Depending on which functions a request will use, the profiled times are looked up in a table and the processing requirements are summed up. In this context, as processing functions, they have primarily considered functions related to security, such as schema validation, encryption/decryption and signature validation. The SLAs with clients define static QoS classes, the requests are classified into. The web server that accepts the client requests forwards all requests to a host that acts as the security processor. A request classification module inspects the HTTP header and SOAP headers of each request and separates them into different queues mapped to the QoS classes. Mismatched packets are dropped. A scheduler module shares the processor according to a pre-defined ratio, using a deficit round-robin algorithm. Most of the aforementioned solutions achieve differentiation between services according to some QoS criteria. As QoS parameters, functional properties (such as response time) and non-functional properties (such as whether a paid customer or not) are both used. Many systems that require predictability in execution hardly benefit from the non-functional QoS parameters. With functional QoS properties such as response time, the aforementioned solutions fail to clearly guarantee predictability of the execution, in their solutions.

Work of Helander J., Sigurdsson S. [13] attempts to use SOAP based web services in an embedded Real-Time environment, where SOAP based web services are used for communication between the different components of an embedded system. Communication patterns between the components are pre-known, together with the timings of execution at each component. The sequence of events that could take place is also previously known, hence they are planned for at design time. When an event takes place, the subsequent events are known and their resource consumptions are worked out. Resources are reserved at each component to make a timely execution. In order to handle variations and jitter in execution, resources are over reserved at each component. A statistical model is used to predict the amount of resources needed. Details about each execution is used to tune the parameters of the system. Due to the closed environment in an embedded system, each communication scenario can be identified and planned for. Predictability is achieved by planning ahead of actual execution to ensure that a task being executed at a particular component has all the resources ready for its needs. This ensures minimal variance in task execution times at each component. Although this guarantees predictability, this solution is not viable for open environments where the requests are not known a priori, hence cannot be planned for.

Extensive amount of research has been done in coming up with a QoS aware middleware based on CORBA by Schmidt D. C. et al. for systems with such stringent QoS requirements. As can be seen in many of their work [4, 19, 20, 21], adherence to stringent QoS requirements is achieved by priority based scheduling and execution in all operations performed for the requests by the Object Request Brokers (ORB) and the subsystems. The fundamental change done to an ORB is to differentiate the operations for each request by introducing priorities. Each subsystem component is designed to ensure end-to-end predictability is maintained by preventing priority inversions in all components. The notion of a priority based thread pool forms the core of the solution. As CORBA implementations are available for many platforms and Operating Systems, the Real-time version is designed to map the thread priorities to the
local Operating System priorities accordingly. A Real-time networking subsystem supports
the thread pool by ensuring that priority based processing by passes certain layers and access
to memory directly. Moreover, techniques such as early demultiplexing and schedule driver
protocol processing ensures the priorities prevail in all its operations. Real-time CORBA is
currently being used in Avionics and Medical equipment. While this is an elegant example for
a Real-time middleware for applications with stringent QoS requirements, the implementation
is only suited for closed systems such as embedded environments. In such systems, the data,
processing, workflows and resource requirements are pre-known and well planned for, offline
at design time. In order to alleviate any problems with jitter and minor processing issues,
the worst case scenarios are planned for with resources being overallocated as compensation.

In web services domain, although the functionality of the services are known to the service
provider, a SOAP engine that hosts them is designed as a product that any web service could
be hosted with. Moreover, the arrival rate and order of client requests and their nature, which
parameters are used as input, what would be execution time requirement as a result, are all
unknown till they are received and executed at the SOAP engine. This results in a more
dynamic and a complex scenario compared to closed systems.

2.2 Summary

Work by various researchers on applying QoS principles to web services was discussed in this
section. There have been many attempts to introduce the notion of a QoS broker either as a
separate entity or by modifying UDDI registries to facilitate the querying of services based on
QoS attributes. Work has also been done on standards and markup languages to facilitate the
description of both QoS levels provided by a service as well as the for consumers to describe
their QoS requirements. Attempts at enforcing QoS at the execution levels has not been
researched by many. Some research done in this direction attempts a broker module that
keeps track of the QoS levels provided by the services hosted and differentiate them based
on pre-negotiated SLAs with clients. One solution categorises requests according to priority
levels and tries to differentiate them by using a Deficit Round Robin type of scheduling. While
these attempts achieve service differentiation to a certain extent, all of them fail to promise
the same level of service everytime the service is invoked. Moreover, how the actual level of
service is achieved in service execution has not been clearly discussed in any of the attempts.
Therefore actually achieving the perceived level of service at the lowest level still remains as
a problem.

3 Background

3.1 Real-Time scheduling algorithms

Real-time systems are characterized by equal importance placed on time taken for a result
to be obtained as on the correctness of the computation performed. If the service time of a
request takes more time than a given deadline, the result obtained can be of little or no use.
Such stringent QoS requirements which are common in Real-time systems, demand the service
execution and the middleware used, to have very high predictability. Scheduling algorithms
in Real-time systems are largely responsible for such high level of predictability. In Real-time
systems, the different types of tasks in a system, their resource requirements, order of arrival
and other such information are pre-known. An assurance of meeting the deadline of each
task is obtained before the system is built. This happens by following certain Schedulability Checks that are worked out offline [22, p. 20].

Web Services, are highly dynamic in nature. Requests made for services hosted might not necessarily be known in advance. A request with Real-time QoS requirements has to be accepted and serviced accordingly, with a guarantee of meeting the requirements. The solution devised, is based on principles that Real-time algorithms are also based on. Some of the well known Real-time Scheduling algorithms and the principles behind them are discussed below.

**Types of Real-time Tasks**

Real-time tasks could be classified according to the nature of their deadline. [23, Ch. 3]

- **Hard Real Time tasks** - Deadlines cannot be missed. Missing it will make the task output unusable and could lead to fatal errors in the system.

- **Soft Real Time tasks** - Missing a deadline will not result the task being unusable, however there will be a penalty involved with it normally.

- **Firm Real Time tasks** - The sooner the tasks finish their computations, the higher the reward is.

Tasks could be further classified according to their frequency of occurrence. [22, 24, 23]

- **Periodic** - Real-time regular arrival times based on a fixed rate.

- **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.

**Real-time Scheduling**

The following assumptions are made about the tasks and scheduling,

- All tasks are hard-real time tasks and they occur periodically.

- Jobs are ready to run at the release time.

- Deadlines are either equal or less than the period.

- Tasks do not suspend by themselves.

- Tasks scheduled are mutually exclusive. A task does not rely on the completion of another.

- The cost in preempting a task is minimum and negligible.

- All task can be preempted anytime.

Earliest Deadline First (EDF) and Rate-Monotonic (RM) are two well known Real-time algorithms that are proved optimal for certain scenarios [25].
Rate Monotonic Algorithm (RM)

- Static in nature - Priorities are assigned (fixed) and order of execution is decided prior to the start of execution.
- Optimal - RMA 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.
- Priorities are assigned based on the frequency of tasks. i.e. Tasks with higher frequencies get higher priorities.
- Schedulable Bound is less than 100%

For a given Task \( \tau_i \), with a worst case execution time is \( C_i \), a period of \( P_i \),

The fraction of CPU time spent in processing \( \tau_i \) is \( C_i / P_i \).

The total CPU time spent in executing \( n \) tasks is:

\[
U = \sum_{i=1}^{n} \frac{C_i}{P_i}
\]

(1)

The worse case schedulable bound \( W_n \) for \( n \) tasks is:

\[
W_n = n(2^{1/n} - 1)
\]

(2)

If \( \sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{1/n} - 1) \), where \( n \) is the number of tasks to be scheduled, then the RM Algorithm will schedule all the tasks to meet their respective deadlines.

RM algorithm works well with Periodic tasks and could be disrupted by aperiodic and sporadic tasks in the system.

Has a complexity of \( O((N + \alpha)^2) \) in the worst case, where \( N \) is the total number of requests in the hyper period of \( n \) periodic tasks in the system and \( \alpha \) is the number of aperiodic tasks in the system.

Earliest Deadline First Algorithm (EDF)

- Dynamic in nature - Priorities are not assigned offline. The order of execution is decided dynamically at the arrival of a task.
- Optimal - 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.
- The task with the nearest deadline for its current request gets the highest priority. When a new task arrives, it is inserted in to the queue of ready tasks ordered by their deadlines.
- Has a higher scheduling overhead than RM as the priorities are computed dynamically.
• Schedulable Bound is 100% for all tasks

For a given Task \( \tau_i \), with a worst case execution time is \( C_i \), a period of \( T_i \).

\[
\sum_{i=1}^{n} \left( \frac{C_i}{T_i} \right) \leq 1
\]  

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 Utilization 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 is more tolerant to Aperiodic and sporadic tasks. However the major problem with EDF is that there is no guarantee which tasks will fail in overload conditions.

Has a complexity of \( O((N + \alpha)^2) \) in the worst case, where \( N \) is the total number of requests in the hyper period of \( n \) periodic tasks in the system and \( \alpha \) is the number of aperiodic tasks in the system.

**Least Laxative First (LLF)**

• The Laxativity of a task is the maximum amount of time the task can wait and still meet its deadline. Laxativity of a process = Deadline - Remaining computation time.

• Dynamic in nature.

• The active job with the smallest laxity gets the highest priority. While a process is executing it can be preempted by another task which has a smaller laxity.

• The problem with the algorithm is that there can be many preemptions happening and the overhead created by it.

• An optimal algorithm for periodic tasks.

• Has a complexity of \( O((N + \alpha)^2) \) in the worst case, where \( N \) is the total number of requests in the hyper period of \( n \) periodic tasks in the system and \( \alpha \) is the number of aperiodic tasks in the system.

Although web service requests could be periodic, the assumption of no prior knowledge of the requests being received, is made devising the solution. The deadline of a task is taken as the primary QoS parameter for scheduling. The envisioned solution is therefore based on Earliest Deadline First principles. This ensures that the solution has a schedulable bound of 100%. However, a schedulability check that supports aperiodic task arrivals is introduced. The check is based on certain principles behind the EDF algorithm, which are outlined below.

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 [26])** *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 [27]) 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 1 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 task \( T1 \) runs, 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.

The optimality of the EDF algorithm is ensured by the schedulability check (depicted in equation 3) it is associated with. With a positive outcome, the check guarantees that all tasks would meet their respective deadlines. As it is meant to be worked offline, it renders itself unsuitable in a web service scenario. However, certain principles behind the check can be accommodated in building a suitable feasibility analysis. Two concepts considered in feasibility analysis, namely \textit{processor demand} and \textit{loading factor} [22] are described here.

A given task \( T_i \), has a release time of \( r_i \), a deadline of \( d_i \) and an execution time requirement of \( C_i \).
**Definition 1** For a given set of real-time tasks and an interval of time \([t_1, t_2]\), the processor demand \((h)\) for the set of tasks in the interval \([t_1, t_2]\) is

\[
h_{[t_1, t_2]} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k. \quad (4)
\]

The total execution time required by all jobs, with release times at or before \(t_1\) and deadline at or before \(t_2\), is calculated as the processor demand in equation 4.

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

\[
u_{(t_1, t_2)} = \frac{h_{[t_1, t_2]}}{t_2 - t_1}. \quad (5)
\]

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

\[
u = \sup_{0 \leq t_1 \leq t_2} u_{(t_1, t_2)}. \quad (6)
\]

As an example, applying these to the tasks in the EDF schedule obtained in Figure 1,

| \(u(0,7)\) | \(\frac{5+3}{7} = \frac{8}{7}\) |
| \(u(0,10)\) | \(\frac{5+3+8}{10} = \frac{16}{10} = \frac{8}{5}\) |
| \(u(5,16)\) | \(\frac{5+3+8+16}{16} = \frac{34}{16} = \frac{17}{8}\) |
| \(u(0,16)\) | \(\frac{5+3+8+16}{16} = \frac{34}{16} = \frac{17}{8}\) |
| \(u(0,20)\) | \(\frac{5+3+8+16}{20} = \frac{34}{20} = \frac{17}{10}\) |

Table 1: Loading factor computation for the job set of EDF schedule in Figure 1

**Theorem 3 (Spuri [28])** Any set of real-time tasks is feasibly scheduled by EDF algorithm if and only if

\[
u \leq 1. \quad (7)
\]

4 Proposed Solution

The envisioned solution, ensures predictability in web services by service differentiation. Compared to other attempts, the solution places a higher emphasis on achieving execution time predictability. Real-time scheduling algorithms are based on certain theorems that ensure tasks could finish by a given deadline. It is impossible to directly apply them in to a Web Services context due to their offline nature. Applying some of the theorems highlighted in the previous section, a model that could cater the dynamic nature of web services is proposed in this section. The unknown nature of tasks before their arrival at the system, is dealt by performing a schedulability check ‘on-the-fly’ basis and rejecting the tasks that cannot be guaranteed in meeting their deadlines. The check involves calculating the processor demand for the requests accepted and for the newly submitted. In turn the loading factor on the
processor is derived from it and made sure the resultant load be always less than or equal to 100%. The check considers the remaining execution time requirement of the requests already accepted by the system and the total execution time requirement for the newly accepted task. Overall, it calculates the total execution time requirement by the tasks accepted and makes sure at any point of time in the future there can be no deadline misses. An algorithm based on the model is also proposed that details the steps in carrying it out. Moreover, the requests are scheduled according to their deadline requirements such that the request with the earliest deadline is executed first. The algorithm for scheduling the requests in the order of their deadlines is discussed under Implementation.

4.1 Constraints on using deadline based scheduling

Direct application of EDF algorithm and it’s schedulability check into web services domain, proves to be difficult for several reasons.

1. EDF and it’s schedulability check are meant to be used for offline calculation.

A schedule could be worked out offline, only if all its constituent parameters are known in advance. While this is possible with systems such as embedded systems, where a finite set of tasks exists with all properties being predictable. In a web service situation, only the services hosted are known. Neither the order of task arrival, the frequency of occurrence nor their execution time requirements are known in advance. Therefore, an online system is mandated.

2. Variance in execution time.

In trivial real-time systems the types of tasks in the system together with all possible executions are known at design time. Although, there could be a range of execution time requirements, they remains static for the the lifetime of the system. In the web service scenario, the execution times cannot be known in advance. It would largely depend on the services that are hosted, the nature of the service and their parameter range.

3. All tasks being Aperiodic

The original definition of EDF algorithm was for Periodic tasks. However, certain derivatives of EDF supports Aperiodic tasks to be scheduled with periodic and sporadic tasks. Hence, a mixture of tasks is considered, where for a portion of them the properties are known in advance and for aperiodic tasks only the execution time requirement is known. When applied to web services, all tasks need to be treated aperiodic due to prior unavailability of information.

4. Sequence of task occurrence not known

In most real-time systems, the sequence of tasks that could occur throughout the lifetime of the system is known in advance. Although, it might not be practical to know the sequence from start of the system continuously, due to possible sporadic or aperiodic events, the order of tasks after a particular event occurs is highly predictable. In web services, the sequence is highly dynamic and depends upon how the clients make requests for the services hosted.
4.2 Features of the solution

The envisioned solution caters to the dynamic nature of requests highlighted above. The basic principles behind the solution are as follows,

1. Deadline based scheduling

   The tasks are scheduled according to their deadline requirement, which is considered as the QoS parameter to meet.

2. Priority based task pre-emption

   SOAP engines service tasks in a best effort manner. For instance, if 100 requests are received by a SOAP engine, it will try to service all of them, as they arrive. The only limitation being the number of threads the engine has active, to service requests. All requests would be serviced and executed in parallel. The threads would share the processor as most SOAP Engines and Operating Systems employ a processor sharing based algorithm. In the devised solution, priority levels are introduced to the system. Requests to be serviced are assigned a priority based on their QoS requirements. A priority based scheduling algorithm is implemented and the scheduler decides which Request should be given the priority to use the processor. This reduces the parallelism in execution to a large extent, however thereby achieving predictability of Request execution.

3. QoS attribute based priority assignment

   In the devised solution, the basis of priority assignment is done on the QoS requirements of the Requests. This happens dynamically as no information about the next request to arrive is available to the scheduler at any given time.

4. Schedulability Check based Task Acceptance

   Whenever a request is received, a Schedulability Check is carried out by the scheduler to ensure that the QoS requirements of the request can be met. If the request can be accommodated, it is accepted, otherwise rejected. Due to the rejection, the client would know that the request cannot be accommodated at the present time. This enables the client to retry the request after a brief time period, if required.

4.3 Proposed Model

The devised model is based on a task Deadline and requires the clients to specify it as a QoS parameter for each request submitted. It is considered as the absolute time period the request has to be serviced within. It is assumed that the users of a web service would have prior knowledge of the average execution times the service would incur. The ways and means of conveying such information to the users, is considered as beyond the scope of this research.

When scheduling requests according to their Deadlines, the problem becomes challenging due to the dynamic nature of requests. As information on requests are not known prior to their arrival, scheduling in-advance or offline, is impossible. Moreover, a schedule that is worked out would not be valid as soon as another request arrives at the system. Hence, the solution needs a way to dynamically change the schedule and ensure that a newly accepted task does not impact the tasks already accepted.
Employing the notion of ‘Earliest Deadline First’, the request with the earliest deadline at a given time is assigned the highest priority in the system and is assigned the processor to run. The priorities are re-assigned at the acceptance of a new task to the system or at the end of a task completion. For instance, with the acceptance of a new task, the task with the highest priority might be preempted if the new task has the earlier deadline. While the scheduling algorithm ensures that the task with the earliest deadline is given the processor, the schedulability check in the solution ensures that a task is accepted only if it’s QoS requirements can be met.

The inherent schedulability check in EDF works well, when calculated offline. Classically, it considers the relative use of the task in obtaining the ratio between it’s execution time and the period. This is summed up for all the tasks in the system, which is trivial when done offline, and checked to see whether the Utilization obtained is less than 100%. The immediate problems with this approach, considering a dynamic scenario is that this considers the total execution time requirement of the task and the periodicity of a task is not known. With a dynamic environment, when the check is done at run-time, the execution of certain tasks in the system would have already begun. Hence, the notion of ‘Remaining Execution Time’ is introduced.

In a pre-emptive scheduling system, a given request’s execution 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 \( e_n \), the Total time of the task execution \( E_i \) can be considered as,

\[
E_i = \sum_{j=1}^{n} (e_j - s_j).
\]

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

\[
R_i = C_i - E_i.
\]

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. First, a segregation of the currently accepted tasks is done, on the basis of whether a task’s deadline lies within the lifetime of the new task or thereafter. First part of the check validates whether the new task’s deadline could be met while ensuring on-time completion of tasks having earlier deadlines.

Let \( T_{\text{new}} \) be a newly submitted task, with a release time of \( r_{\text{new}} \) and a deadline of \( d_{\text{new}} \) and an execution time requirement of \( C_{\text{new}} \). Let \( P \) be the set of tasks already accepted and active in the system, with their deadlines denoted as \( d_p \).
With reference to equation 4, the processor demand within the duration of the newly submitted task can be defined as,

\[ h(r_{\text{new}}, d_{\text{new}}) = \sum_{r_{\text{new}} \leq d_p \leq d_{\text{new}}} R_p + C_{\text{new}}. \] (10)

With reference to equation 5, the loading factor within the duration of the newly submitted task can be defined as,

\[ u(r_{\text{new}}, d_{\text{new}}) = \frac{h(r_{\text{new}}, d_{\text{new}})}{d_{\text{new}} - r_{\text{new}}}. \] (11)

With 11, if the following condition is satisfied, the new task is considered schedulable together with tasks finishing on or before it’s deadline, with no impact on their deadlines.

\[ u(r_{\text{new}}, d_{\text{new}}) \leq 1 \] (12)

With the above condition 12 satisfied, the task is checked for schedulability with the tasks finishing subsequently. Unlike 10, the calculation of processor demand needs to be done for each task with deadlines after \( d_{\text{new}} \), separately. Let \( Q \) be the set of tasks already accepted and active in the system, required to finish after \( d_{\text{new}} \) (such that, with deadlines after \( d_{\text{new}} \)). Let \( q \) be the member of \( Q \), with a deadline of \( d_q \) up to which the processor demand is calculated for,

\[ h(r_{\text{new}}, d_q) = h(r_{\text{new}}, d_{\text{new}}) + \sum_{d_{\text{new}} \leq d_i \leq d_q} R_i. \] (13)

The result of 10 is used as part of the equation. This represents the processor demand of all tasks finishing on or prior to \( d_{\text{new}} \) and can be treated as one big task with a release time \( r_{\text{new}} \) and a deadline of \( d_{\text{new}} \) respectively. Next, the loading factor for the same duration is calculated.

\[ u(r_{\text{new}}, d_q) = \frac{h(r_{\text{new}}, d_q)}{d_q - r_{\text{new}}}. \] (14)

The loading factor is also calculated in 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.

\[ u(r_{\text{new}}, d_q) \leq 1 \] (15)

As this check is repeated for each member of \( Q \), it would be logical to consider the members of \( Q \) in the increasing order of their deadlines. This ensures that the failure of a condition happens as early as possible and prevents the check for the rest of the tasks, in the case of a failure. In summary, for a newly submitted task to be accepted to the system, condition 12 needs to be satisfied for tasks with deadlines on or before \( d_{\text{new}} \), subsequently condition 15 needs to be satisfied separately, for each task with deadlines after \( d_{\text{new}} \).
4.4 Proposed Algorithm

Based on the above model, the following algorithm is derived, which will be the core of the solution in the implementation that follows. Until a web service is invoked, there is no knowledge of the actual execution time, the invocation would take. Moreover, it could vary according to the parameters used for the invocation. When a request is checked for schedulability, the model requires the execution time to be available for the computations highlighted in 4.3. It is assumed that the execution times of each service hosted, is available to the server.
Algorithm 1: Algorithm for Schedulability Check

input: \((S_{new})\) Current time, \((D_{new})\) deadline for the new request \(N\), \((T)\) List of requests currently in the system

output: true: if the task can be scheduled, false: if the schedulability check fails

1. \(WPD \leftarrow 0; \ APD \leftarrow 0\)
2. \(P \leftarrow \text{GetTasksFinishingWithinNewTask}(T, S_{new}, D_{new})\)

3. foreach \(P' \in P\) do
   4. if execution information for \(P'\) exists then
      5. \(WPD \leftarrow WPD + \text{GetRemainingTime}(P')\)
   6. else
      7. \(WPD \leftarrow WPD + \text{GetExecutionTime}(P')\)
   8. end

9. end

10. \(WPD \leftarrow WPD + \text{GetExecutionTime}(N)\)

11. \(\text{LoadingFactor} \leftarrow \frac{WPD}{D_{new} - S_{new}}\)

12. if \(\text{LoadingFactor} > 1\) then
    13. return false
14. end

15. \(Q \leftarrow \text{GetTasksFinishingAfterNewTask}(T, D_{new})\)
16. \(Q \leftarrow \text{SortByDeadline}(Q)\)

17. foreach \(Q' \in Q\) do
   18. \(DL \leftarrow \text{GetDeadline}(Q')\)
   19. \(R \leftarrow \text{GetTasksFinishingBetween}(T, D_{new}, DL)\)

20. foreach \(R' \in R\) do
   21. if execution information for \(R'\) exists then
      22. \(APD \leftarrow APD + \text{GetRemainingTime}(R')\)
   23. else
      24. \(APD \leftarrow APD + \text{GetExecutionTime}(R')\)
   25. end

26. end

27. if execution information for \(Q'\) exists then
   28. \(APD \leftarrow APD + WPD + \text{GetRemainingTime}(Q')\)
   29. else
      30. \(APD \leftarrow APD + WPD + \text{GetExecutionTime}(Q')\)
   31. end

32. \(\text{LoadingFactor} \leftarrow \frac{APD}{DL - S_{new}}\)

33. if \(\text{LoadingFactor} > 1\) then
    34. return false
35. end
36. end
37. return true
The algorithm is used to carry out a Schedulability Check before any request is accepted for execution. Hence, the logic is tested each time a request is received by the system. The algorithm takes the current time, the deadline of the new request and the list of requests already accepted by the system’s inputs. Current time serves as the start time of the request. As per the model described in 4.3, The check consists of two steps. Firstly, requests having deadlines within the lifespan of the new task is selected (line 2). First part of the check ensures that the new request could be scheduled together with tasks finishing within it’s lifespan, while meeting all deadlines. For each request $P' \in P$, it is checked whether execution information is currently available (line 4). If the request has been partially processed, the remaining execution time calculated as per equation 5 is obtained (line 5). If the request is yet to be processed, the execution time requirement for the task (line 7), is used alternatively.

In accordance with equation 10, the processor demand within the duration of the newly submitted request is calculated by summing up the remaining execution times or execution time requirements of each task. Subsequently, adding the execution time requirement for the newly submitted request (line 10) completes the processor demand calculation for its lifespan. As per equation 11, the loading factor for the time period is calculated (line 11). If the loading factor is greater than 1, the request is straightaway rejected. If the loading factor remains less than 100% the check continues on to the second stage.

The second stage of the check validates the effect of the newly submitted request on requests finishing thereafter. The requests with deadlines later than that of the newly submitted ,are selected out of the list (line 15). The second stage check mandates it to be done separately for each and every request selected. To make the process more efficient, the selected list of requests are first sorted in the ascending order of the deadlines (line 16). For each request $Q' \in Q$, a further subset of requests from the list selected. All requests required to finish between newly submitted and $Q'$ are selected into set R (line 19). For each request $R' \in R$, the processor demand is calculated by using either the remaining time of the request or it’s execution time requirement (lines 21 to 26). To this, the remaining time or the execution time request of request $Q'$ is also added (line 27 to 29). As per equation 13, the processor demand calculated for the duration of the new request is added to it (line 28 and 30). This step is analogous to treating all requests (including the newly arrived) finishing on or before the new request, as a single unit. Finally with reference to 14, the loading factor is calculated for the same time period (line 32). If the resulting loading factor exceeds 100%, the request is rejected (line 33 to 35). If it is less than or equal to 100%, the same check is done for subsequent requests in $Q$, until a check fails or all of them are satisfied. At that point, the request is considered schedulable.

Sorting the requests by ascending deadlines ensures that if a check fails, it happens as early as possible. Moreover, it avoids the check being done for subsequent tasks in the set, after a failure. Depending on the sorting algorithm used, the algorithm’s time complexity could vary. The complexity of the algorithm excluding the sorting function calculates to be $O(n^2)$. If the sorting function used is either bubble, insertion, selection, quick or shell sort, the resulting complexity would be $O(n^2)$. If the sorting function is heap or merge sort then the sorting function would have a time complexity of $O(n)$, nevertheless resulting in $O(n^2)$ due to the higher complexity of rest of the algorithm.
4.5 Sample Scenario

The following example illustrates the functionality of the algorithm. Each step identifies the arrival of a request to the system and the schedulability check performed on it. Properties of the requests, their arrival times, execution requirements are summarized in the Table 2, in the order of arrival.

<table>
<thead>
<tr>
<th>Request</th>
<th>Start Time (ms)</th>
<th>Execution Time (ms)</th>
<th>Deadline (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>T2</td>
<td>1</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>T3</td>
<td>3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>T4</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>T5</td>
<td>7</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>T6</td>
<td>8</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>T7</td>
<td>9</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2: Properties of Requests

The system starts with no requests being serviced. The arrival of the first request $T1$ is shown in Figure 2. As per Table 2, $T1$ has an execution time requirement of 5ms and the execution needs to finish within a deadline of 25ms. The remaining execution time is illustrated using a dotted line in Figure 2, while the deadline has been marked using a straight line. As this is the first request arriving at the system, the schedulability check is not done for $T1$.

Request $T2$ arrives in the system 1ms later. By 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 2 - 9) is not applicable. However, rest of the check (lines 10 - 37) is applied as follows.
Figure 3: Arrival of Request T2

Proc. Demand Within = (0 + 6)ms (Algorithm 1: line 10)
= 6ms

Proc. Demand After = (0 + 4)ms (Algorithm 1: lines 20 - 26)
= 4ms

Total Proc. Demand = (6 + 4)ms (Algorithm 1: lines 27 - 31)
= 10ms

Loading Factor = \( \frac{10}{\frac{25}{1}} \) (Algorithm 1: line 32)
= 0.4167

0.4167 > 1 (Evaluates to False - Accept request)

As can be seen in the calculation above, the processor demand between the arrival time of T2 (or current time) and the deadline of T1 is calculated. This constitutes 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 100%. Hence, T2 is accepted for execution. With T2 now being the request with the earliest deadline, the processor is immediately obtained by it for execution. As illustrated in Figure 3, remaining execution of T1 is delayed till execution of T2 finishes. However, T1 can finish within its deadline.

2ms into the execution of T2, T3 arrives at the system and the schedulability check is performed on it. Similarly to T2, there are no requests in the system with deadlines prior to that of T3. Hence, lines 2 - 9 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 (Algorithm 1: lines 27 - 31) 
= 8ms

Loading Factor = \[
\frac{8}{(20-3)}
\] (Algorithm 1: line 32) 
= 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 = \[
\frac{12}{(25-3)}
\] (Algorithm 1: line 32) 
= 0.545

0.545 > 1 (Evaluates to False - Accept request)

The sorting function in Algorithm 1: line 16 results in the requests being ordered according to ascending 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 100%. 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 54.5%, hence the task is accepted as illustrated in Figure 4.

With its acceptance, T3 preempts T2 to claim the processor as the task with the earliest deadline. As illustrated in Figure 4, T2 will recommence execution after 4ms followed by T1 recommencing execution after another 4ms. Although T2’s execution is staged and the recommencement of T1 further delayed, the resulting schedule ensures all 3 requests meeting their respective deadlines.

![Figure 5: Arrival of Request T4](image)

Request T4 arrives at the system 1ms into the execution of T3. 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.

\[
\text{Proc. Demand Within} = (0 + 2)\text{ms} \quad \text{(Algorithm 1: lines 2 - 9)} \\
= 2\text{ms}
\]

\[
\text{Total Proc. Demand up to T4} = (2 + 4)\text{ms} \quad \text{(Algorithm 1: line 10)} \\
= 6\text{ms}
\]

\[
\text{Loading Factor} = \frac{6}{(11-4)} \quad \text{(Algorithm 1: line 11)} \\
= 0.86
\]

\[
0.86 > 1 \quad \text{(Evaluates to False)}
\]

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 (Algorithm 1: lines 27 - 31)
= 10ms

Loading Factor = \frac{10}{20 - 4} (Algorithm 1: line 32)
= 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} (Algorithm 1: line 32)
= 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. When Loading Factor is calculated for the time period, it results to be less than 100% indicating no deadline misses. This leads the check to continue on to the second part where tasks finishing after T4 is considered.

Following the same steps as before, a sorted list of requests with deadlines after that of T4 is obtained. 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 100% the same is calculated for the time period between deadline of T4 and the deadline of T1. As the loading factor calculates to be less than a 100% for the time period, T4 is accepted for execution.

Since T3 still has the earliest deadline out of all tasks in the system, it continues to have the CPU for execution. However, at the completion of T3, T4 the request with the next earliest deadline will get the CPU for execution. This results in the execution of T2 and T1 being further delayed. As illustrated in Figure 5, 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. Although the execution of T2 and T1 has been further delayed, both requests still manage to finish execution within their respective deadlines.

T5 is a relatively small task arriving at the system 1ms into the execution of T4. Moreover, it has a relatively small deadline with a buffer of 1ms. With its arrival the schedulability check is carried out. As T5 has a deadline earlier than the rest of the requests, the first part of the check is skipped.
Processor demand is first calculated for the time period between the deadline of T5 and the deadline of T4 as the sorting function results in 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, this results in a loading factor of 1.25 which fails the test. Hence the request T5 has to be rejected.

A loading factor 125% 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. Figure 6 illustrates how the executions would take place in such a scenario. 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. As can be seen in Figure 6, T5 would execute from 7th - 9th millisecond, after which T4 would resume execution. As T4 has a remaining execution time of 3ms, it results in T4 completing execution only at the 12th millisecond in the timeline. 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.

**Figure 6: Arrival of Request T5**

![Figure 6: Arrival of Request T5](image-url)
Figure 7: Arrival of Request T6

After the rejection of T5, request T6 arrives at the system 2ms into the execution of T4. As T6 has a deadline after that of T4, the entire schedulability check is applied to it.

\[
\text{Proc. Demand Within} = (0 + 2)\text{ms} = 2\text{ms}
\]
\[
\text{Total Proc. Demand up to T6} = (2 + 7)\text{ms} = 9\text{ms}
\]
\[
\text{Loading Factor} = \frac{9}{(18-8)} = 0.9
\]
\[
0.9 > 1 \quad \text{(Evaluates to False)}
\]

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

\[
\text{Proc. Demand up to T2} = (0 + 4)\text{ms} = 4\text{ms}
\]
\[
\text{Total Proc. Demand} = (9 + 4)\text{ms} = 13\text{ms}
\]
\[
\text{Loading Factor} = \frac{13}{(20-8)} = 1.083
\]
\[
1.083 > 1 \quad \text{(Evaluates to True - Reject Request)}
\]

Unlike T5, T6 successfully goes through the first part of the check. A loading factor of 90% within the lifespan of T6 means that it could be successfully scheduled to meet its deadline while the already accepted T4 having a deadline earlier than that of T6, meets its deadline successfully as well. 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. Due to T6 being a considerably large task with a tight deadline (with a buffer of just 3ms), the
resultant processor demand and subsequently the loading factor for the time period results in 108% of CPU utilization. This leads to request T6 being rejected.

Figure 7 illustrates the scenario of accepting T6 for execution. Since the loading factor was less than 100% within the lifespan of T6, both T4 and T6 would be able to meet their respective deadlines. T6 having a deadline earlier than that of both T2 and T1, causes both those request to further wait for the CPU to resume execution. On the timeline, T6 would start execution at the 10th millisecond, right after T4 finishes execution and continue owning the CPU till the 17th millisecond. This results in T2 resuming execution at the 17th millisecond and running for 4ms followed by T1. As can be seen on Figure 7, T2 would miss its deadline by 1ms as a result of being pushed back. Although T1 may still meet it’s deadline, the fact that T2 misses its deadline means T6 would have a negative impact on at least one of the tasks already accepted. Hence, the rejection of T6 is justified.

\[
\begin{align*}
\text{Proc. Demand Within} & = (0 + 1) \text{ms} \\
& = 1 \text{ms} \\
\text{Total Proc. Demand up to T6} & = (1 + 2) \text{ms} \\
& = 3 \text{ms} \\
\text{Loading Factor} & = \frac{3}{(15-9)} \\
& = 0.5 \\
0.5 & > 1 \quad \text{(Evaluates to False)}
\end{align*}
\]

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 = (3 + 4)ms  
= 7ms  

Loading Factor = \( \frac{7}{(20-9)} \)  
= 0.636  

0.636 > 1  \((\text{Evaluates to False - continue to next check})\)  

Proc. Demand up to T1 = (4 + 4)ms  
= 8ms  

Total Proc. Demand = (7 + 8)ms  
= 15ms  

Loading Factor = \( \frac{15}{(25-9)} \)  
= 0.9375  

0.9375 > 1  \((\text{Evaluates to False - Request accepted})\)

With T4 having only 1ms of execution time remaining, the processor demand between the lifespan of the newly arrived T7 accumulates up to a minute 3ms. This could be comfortably scheduled within the 6ms lifespan of T7. Therefore the first part of the schedulability check results positive. The time duration between the deadline of T7 and that of T2 yields a processor demand of 7ms to be completed within a time period of 16ms. Therefore, this results in a loading factor that is less than 100%, being a valid schedule. Thereafter, the processor demand between the deadline of T7 and that of T1 is calculated. This results in a total of 15ms of execution time to be scheduled within an interval of 16ms. The resultant loading factor of 93.75%, indicates that the request T7 can be successfully scheduled with each of the already accepted tasks executing in the system.

![Completed Schedule of all accepted Requests](image)

Figure 9: Completed Schedule of all accepted Requests
Figure 9 illustrates the completed schedule for all accepted requests. Although some tasks were executed in a staged manner, all accepted tasks were able to meet their deadlines successfully. Furthermore, it can clearly be seen that the deadline / execution time ratio and the arrival time plays a part in which request gets accepted and which is rejected. Rejection of requests with deadline and execution time requirements that cannot be accommodated within the available CPU time, ensures that already accepted requests in the system would not be penalized for their execution.

5 Implementation

The proposed model and algorithm is tested for practical use by augmenting an existing widely used SOAP engine. The expectation here is to provide predictability and achieve service differentiation in the services hosted. This section contains a brief introduction to the different ways of developing web services followed by details of the architecture of Apache Axis2, the open source product used to implement the solution. Furthermore, the detailed implementation aspects and challenges faced are also discussed in this section.

5.1 Development and Deployment of Web Services

A Remote Procedure Call, in theory involves steps to pack the parameters used by the remote procedure before being sent across the network and unpacking them on the remote server after being received. These steps are referred to as parameter marshalling. When web services are the middleware being used, similar steps are a necessity. This involves the serializing/deserializing of data, translation of programming language types to XML SOAP types and vice versa. To maintain location transparency, these steps are carried out by either the web service libraries or by a SOAP Engine application.

There are many ways and technologies used to develop Web Services. Languages and development platforms contain toolkits such as JAX-WS[29] part of the Java development platform from Sun Microsystems, Windows Communications Foundation[30] from Microsoft, which allows developers to model entities and services adhering to business and programming principles or reuse code already written with minimum or no modifications. The libraries take care of the housekeeping activities such as serializing/deserializing data, parameter marshalling and mapping of SOAP XML data types to language types. Similarly, there are multiple ways to deploy a web service. The simplest form is for the web service libraries to create hooks to an existing web server such as Apache[31] and expose the services as a services. gSOAP Web Services Toolkit [32] allows the services to simply be deployed as Common Gateway Interface (CGI) applications. Using the same, alternatively the services could also be deployed through a standalone daemon. Using a JAX-WS compliant implementation allows the services to be deployed in several application servers.

An application server is designed to host, execute and manage components such as Enterprise Java Beans (EJB)[33], providing them with thread and resource support. A web service to be deployed, will be modelled conforming to EJB specification or the code generated by the libraries used, will encompass the written web service to conform to widely accepted component specifications. These application servers are designed to host multiple services. Application servers such as Apache Axis2[34], Glassfish [35], JBoss[36] are widely known and referred to as SOAP Engines.
It was decided to use a SOAP engine based approach due to its ability of hosting multiple services, and Apache Axis2[34] was selected as the preferred choice due to its wide use and availability in both Java and C language versions.

5.2 Client QoS Requirements

The service differentiation is based on certain QoS parameters requested by the web service consumers. With each web service call made, the clients will specify the following parameters to be used by the scheduling algorithms on the server. In the context of the current solution, the Deadline or the time duration within which the client perceives the web service call to end is specified as a QoS parameter. This could also be seen as the worst case execution time. These parameters are conveyed to the server using SOAP headers. Figure 10 illustrates a SOAP packet containing QoS parameters, from an actual call made. It is assumed that the programmers authoring clients to use a web service would have some knowledge about the service and would indicate the perceived deadline as part of the web service call.

5.3 System components

For applications with real-time requirements, predictability of execution is the most important factor. All attempts discussed under related work fails to achieve a guaranteed level of predictability. In order to achieve the perceived level of predictability, it must be supported by every level of the system. This includes not only the devised solution by design but the development platform, libraries used and the operating system as well. In order to get the required predictability at the operating system level, a real-time operating system must be used. Sun Solaris version 10 08/05 and Linux with the real-time kernel were selected for the server and client machines respectively. While any Linux distribution with the real-time kernel would have been adequate, Ubuntu 8.10 with the real-time kernel was selected for the ease of installation and operation.

Although Apache Axis2 is available both in Java and C versions, the Java version is it’s flagship product. Using the Java version would also cut down the time on understanding the code base as well as on implementing any enhancements in the product. However, Java as a language does not have any real-time capabilities. Java by design is a strongly typed language that employs a Garbage Collector (GC) mechanism to manage memory. As a
result when memory is to be reclaimed, the GC is run interrupting all other threads of execution. Furthermore, Java is treated as an interpreted language where a ‘runtime’ is needed to interpret the Java byte code which source code is compiled to. Java employs on-demand compilation (known as Just In Time compilation) in order to conserve processing resources. All these features of Java have made it unsuitable for Real-time system development.

With the advent of Real-Time Java Specification [37] (RTSJ) all these issues have been dealt with by design. RTSJ introduces Real-Time threads that can be assigned with higher priority levels than the normal Java threads. Furthermore, these new priorities include levels that cannot be interrupted by the GC and the ability to pre-load classes to prevent the overhead associated with Just-In-Time techniques. Sun Java Real-Time System version 2.1 was used for this implementation.

5.4 Apache Axis2 Architecture

A SOAP engine’s functionality can be broadly defined as serving web service requests acting both as a manager as well as a container. Although there are many SOAP engines available all of them share certain common components. The details of the architecture of a generic SOAP engine could be found in Appendix 9. The functionality represented by the various modules can be seen on any SOAP engine regardless of the granularity which they could be differentiated at, in the actual implementation. However, different SOAP engines may have additional modules that help in its functionality and may use various techniques to achieve efficiency.

Apache Axis2 is one of the most popular and widely used open source SOAP engine available. It has been built to support both SOAP based and RESTful web services and is highly extensible in nature. As can be seen by Figures 11 & 12, the Apache Axis2 SOAP engine architecture is a very specialised architecture. One of the primary goals of its architecture is the ability for anyone to easily extend its functionality. The architecture has a higher level module structure that are known as ‘phases’. When SOAP messages are processed, the execution order of these phases is pre-defined (the order can be configure by developers). Each phase can have one or more ‘handlers’. Handlers are synonymous to modules that perform

![Apache Axis2 Architecture](image-url)

Figure 11: Apache Axis2 Architecture: Outgoing message processing
a particular task on a SOAP message or on the ‘message context’ which is the internal data structure in Axis2 that envelopes a SOAP message.

While pre-defined handlers are provided out-of-the-box, developers can easily add, remove, modify or extend the existing handlers. Each phase plays a distinct role in SOAP message processing. Therefore the handlers that are grouped into a phase, perform tasks that are synonymous to that phase. The handlers in each phase are executed in a particular order which can also be defined by the developer.

Figure 13 illustrates some of the internal components of Axis2. Although having different names, the modules of a Generic SOAP engine described is clearly visible from it and the throughout the different phases.

Message processing functionality of the Axis2 engine is based around a thread pool that services all incoming requests using a set of worker threads. By default it operates in a best effort scenario where the threads are not differentiated by any means. This means that all
the ‘active’ threads in the pool share the processor at any given time. They have an equal opportunity to get at the processor and as a result, the more threads that share the processor the longer each of the thread executions would take. In the architecture of Axis2, a worker thread would be assigned to carry out the processing of a single request at any given time. In the thread’s execution the request would be transitioning through the different components inside the SOAP engine, each carrying out a unique task. Once the request processing is completed and depending on the message exchange pattern, a reply is sent to the client, the thread returns back to the pool. Therefore the lifespan of a request inside Axis2 is spent always with a single worker thread.

Upon receiving a request, an internal information model is used to envelope the incoming SOAP message and gets passed along the processing path from one module to another. This composite data structure is known as a Configuration Context in Axis2 and additional information gets filled up into it as the processing takes place in different modules. Figure 14 illustrates the multi-hierarchical nature of the Configuration Context. The Context hierarchy stores information that is dynamic in nature and which are generated as the request and the messages are processed by the different components. Whereas, the Description hierarchy represents static data that is read from configuration files and these do not change in the lifetime of Axis2. As the names themselves imply each level of the hierarchy represents information at Global, ServiceGroup, Service, Operation and Message level respectively. The information is stored as key-value pairs and when searched for at any level, the search will move up the hierarchy until a match is made. All operations in Axis2 are stateless.

5.5 Enhancements made to Axis2

In implementing the model discussed in section 4, the following enhancements are made to Axis2. After understanding the internals of Axis2, the enhancements were done at multiple
places that is most suited for the respective desired outcome. Figure 15 illustrates Axis2 components that have been changed in implementing the solution. The Listener Manager and Request Executor thread pools have been replaced with custom made Real-time thread pools. Moreover, a Real-time scheduler component has been introduced to ensure that the threads are scheduled according to the Real-time algorithm used. Enhancements to the SOAP processing module enables communication of QoS requirements of the client, using SOAP headers. The Dispatcher module is modified to support the use of Real-time threads and to enable custom logging that is used for performance measurements. Finally, modifications to the information model is done to exchange data structures and information between the different components and sessions. The rest of the Axis2 components remain largely unchanged.

5.5.1 Priority based thread model

Although Axis2 employs a thread pool in its functionality, all worker threads executing are treated at the same priority. As a result, all active threads will share the processor in execution. To enable service differentiation, a priority based thread model is introduced into Axis2. Which worker thread is given the processor for execution is decided at runtime and it is controlled by changing the priorities assigned to the worker threads. The extended priority model in RTSJ is used for this purpose. Whenever the priority of a thread is set to a higher priority than the others, it is automatically given the processor for execution, exclusively. Similarly, the currently executing thread can be pre-empted by assigning a lower priority than the others. If more than one thread has the highest priority in the system at a given time, all threads with that priority would share the processor automatically. The decision of which thread gets which priority, is decided according to the dynamic scheduling algorithm used. Primarily 4 priority levels are used in the implementation.

1. **Low priority** - is set to one priority level higher than the highest level available for regular java threads. This is to ensure that it they will get a higher priority than a normal thread in the system, especially when priorities are handled at the Operating system level.

2. **Mid priority** - is set to the mid range of the new priority levels in RTSJ. However, threads with mid priority can be preempted by the GC.
3. **High priority** - is set at two priority levels lower than the highest available priority level in RTSJ. This would be the default priority a worker thread is assigned to when it is meant to be in execution.

4. **Highest priority** - is set to the highest available priority in RTSJ, is also used. This level is primarily used to prevent a processor sharing mode in the event of a higher priority thread promoting the priority of another thread. This happens when the scheduler reschedules all threads.

### 5.5.2 Real-time Thread pool design

At the heart of Axis2 is the thread pool that services all web service requests submitted. Currently it makes use of a standard thread pool available with Java libraries. The introduced priority based thread model is supported by a specialised implementation of a thread pool that makes use of Java Real-time threads. The thread pool can be custom initialized on creation, by specifying the maximum number of threads and the starting number of threads. This allows us to pre-create threads at start to avoid the overhead of thread creation.

Axis2 employs two thread pools in its functionality. One is used as a listener manager listening for requests on a pre-defined port. This pool has a single worker thread. The second pool is used for the execution of requests. By design, a worker thread is handed the incoming connection and the worker thread takes the request through all the phases until it is completed. Both thread pools are replaced by the specialised thread pool in order to enable real-time priorities. The listener manager worker thread executes at high priority being statically set at. It is to ensure that the listener does not miss out on any requests as a result of the thread being preempted. Worker threads on the execution pool start at a default high priority. A thread-safe queue is used to dispatch requests to worker threads. Pre-created worker threads blocks on the request queue until submissions are done.

The submission of task triggers a waiting thread to pick it up and start the execution. At submission, if all threads are occupied, the pool will create a new thread if the maximum number of threads is yet to be reached. Upon reaching the maximum, subsequent tasks are blocked on the request queue waiting to be picked up by the next free thread. A worker thread upon finishing a task will service any waiting requests on the queue. If the task queue is empty and minimum number of threads in the pool has been reached, the thread simply dies off. If the minimum number is not reached, the thread gets blocked on the task queue waiting for a task.

Figure 16 illustrates the design of the thread pool and its supporting classes used for scheduling. The core functionality of the thread pool is implemented within the `RTThreadPoolExecutor` class. By implementing the standard `java.util.concurrent.ExecutorService` interface, the custom threadpool automatically achieves compatibility with the rest of Axis2 code designed to use a thread pool. The request queue is implemented using a thread safe `java.util.concurrent.LinkedBlockingQueue` instance. It is an unbounded queue made up of linked nodes arranged in a First-In-First-Out (FIFO) order. Due to this property the LinkedBlockingQueue performs insertions and removals in a time complexity of $O(1)$. The Worker threads are kept track of using a thread safe `java.util.LinkedList`. With an insertion time complexity of $O(1)$ and search time complexity of $O(n)$, this proves ideal as threads are accessed regularly as batches depending on their status.

Worker threads are modelled using `RTWorkerThread` instances which subclass Java `Real-
timeThread, the Realtime version of Java Threads. This enables the use of additional priorities that ensure they are uninterruptable by the GC. Each thread is assigned a state depending on whether it has a request assigned for execution. A worker thread with a request assigned to it would be marked ‘Active’, while one without a request attached is set as ‘Inactive’. This is to enable the selection of Threads with requests only for rescheduling.

Each submitted request is wrapped in a RTTask instance, which stores Task specific information that is used by mainly the Scheduling Algorithms. RTTask is implemented as a subclass of java.langRunnable and the mandatory implementation of the run() method internally calls the same method on the request object (java.langRunnable instance) it aggregates. RTTask maintains the execution state of the request it encompasses, at two levels. A task status attribute in it is set to Submitted as soon as the request arrives at the system. Once the

![Real-time SOAP - Thread Pool Class Diagram](image)

Figure 16: Real-time SOAP - Thread Pool Class Diagram
request is accepted for execution, it is changed to Validated. The stage of execution a task is at, is kept track of at a more granular level. An attribute is set to one of (Pre Message Fetch, Message Fetched, Executed, Reply Sent and Cleaning Up) stages depending on the current stage a given task is at.

Each execution of a task is measured and information about the execution times are stored as an average per groups of input values, as instances of RTExecutionInfo class. The number of groupings determines the fine grain level the execution time averages are stored at. A Hashtable is used to store this information in the thread pool. With an insertion time complexity of $O(n)$ and an access time complexity of $O(1)$, this proves to be the most suited as insertions are done only once per group value. However, searching the values happen each time an execution is done and due to ‘by reference’ nature of objects, the instance can be directly updated with the newly computed average. The insertion time is optimized to a near $O(1)$ by pre-initialising the Hashtable with the number of entries or groups. Each entry is identified by a String based key which is usually made up of the parameters that makes up the groups.

The thread pool makes use of a real-time scheduling algorithm to decide the priorities of each worker thread. It encapsulates an object instance implementing the RTScheduling interface for this purpose.

5.6 Implementing the proposed Real-time scheduling algorithm

Any Real-time algorithm must implement the RTScheduling interface. This decouples the real-time algorithm implementation from that of the thread pool. The RTScheduling interface has two methods signatures, which would carry concrete implementations in the algorithm classes. One of them is to initiate a rescheduling procedure and the other is to ensure there’s a schedulability check prior to the acceptance of a task for actual execution. The initiation of the rescheduling procedure can be triggered by two events. It is called when a new task is ready to be scheduled and when the currently executing task finishes execution. A new task is considered ready to be scheduled, not when it is first submitted to the thread pool, but when the QoS parameters that it should run under is available.

Due to the nature of Axis2 implementation, the SOAP headers are read sometime after the request is assigned to be processed by a worker thread. Therefore, soon after the QoS Parameters are read, all threads need to be rescheduled to make sure the proper thread is given the highest priority to run. Furthermore, this step happens after the schedulability check which ascertains that the requested QoS parameters could be met, given the currently running and already accepted requests.

At compile time, the RTThreadExecutor would be initialised with an instance of a class implementing the RTScheduling interface. The class RTEDFScheduler contains an implementation of the model and algorithm discussed in Section 4. Moreover, it contains the logic to select the request with the earliest deadline at the time and set the priorities accordingly for it to get the processor for execution. Priority assignment to worker threads is implemented through a number of stages.

Pre-scheduling phase

Algorithm 2 illustrates the first phase of priority selection in the pre-scheduling phase. At the beginning of a worker thread’s execution, a web service request ($R' \in R$) is identified to
be either HTTP GET, HTTP POST or a REST style [39] web request (line 1). An example of a request using HTTP GET is a request for a WSDL document of a service hosted. When a request is classified to be a non-HTTP POST request, the current thread’s priority is automatically demoted to a Lower priority (line 11). When the request is a HTTP POST request, the high priority is retained and the execution is carried forward (lines 2 to 9). As the next step the QoS parameters are read from the SOAP header and set for the current thread (lines 2 to 4). The schedulability check is called for, to make sure that the QoS parameters for the new task can be successfully met (line 5). If the schedulability check fails, the client is replied back with the negative response (line 8). If the check returns positive, the scheduler component is alerted and a request is made to re-evaluate the thread priorities (line 6).

The time complexity of the algorithm is governed by invoking the schedulability check on the current thread, which results in $O(n^2)$. The method `rescheduleAllThreads` has a time complexity of $O(n)$.

### Schedulability Check

The schedulability check implementation is directly based on the algorithm discussed in Section 4. Wherever execution information needs to be obtained, it is done by querying for the proper `RTExecutionInfo` instance, depending on the input parameters. Execution time details are maintained from previous runs for each web method per service per parameter range. The schedulability check makes use of this information only if the particular request is yet to start execution. For a request that has already started, it uses the remaining execution time. The first time a web method gets a request, the previous execution information will not be available for the given method and parameter set. Hence, the execution time obtained from profiling runs is used in place of the actual time. Profiling runs are done offline, enabling the server to guess a realistic figure for the execution time.

If the schedulability check for the newly submitted request returns negative, the request is rejected. From the schedulability check code, an `AxisFault` exception with a specific message is raised to indicate the task rejection. Existing Axis2 code is being made use of, in propagating
the AxisFault to the client. If the schedulability check returns positive, the new request is accepted for execution and a reschedule of all threads is called for by the scheduler. Similarly, a request ending execution would trigger a reschedule of all threads.

**Algorithm 3**: Priority shifting of threads based on earliest deadline

```plaintext
input : List of worker Threads T
output: Priority set on each thread

1. earliestThread ← ∅;
2. earliestDL ← ∅;
3. foreach T' ∈ T do
   4. S ← T'.getState();
   5. RT ← T'.getRunnableTask();
   6. if RT is not ∅ and S is Active then
      7. ST ← RT.getStatus();
      8. if ST is Validated then
         9. nextDL ← RT.getDeadline();
         10. if earliestThread is ∅ then
              11. earliestThread ← T';
              12. earliestDL ← nextDL;
         13. else
              14. if nextDL < earliestDL then
                  15. earliestThread ← T'; earliestDL ← nextDL;
               end
         end
      16. T'.setPriority(Mid);
   end
22. if earliestThread is not ∅ then
   23. currentThread.setPriority(Highest);
   24. earliestThread.setPriority(High);
   25. if earliestThread is not currentThread then
      26. currentThread.setPriority(Mid);
   end
28. end
```

**Deadline based scheduling**

`RTThreadPoolExecutor` encapsulates a Real-time scheduler instance implementing the RTScheduling interface. Apart from carrying out the schedulability check based on the specialised implementation, it schedules threads based on the QoS requirements they are tagged with. In this implementation of Deadline based scheduling, the `RTEDFScheduler` instance contains an implementation of Algorithm 3. This code get executed by two events. When a new task passes the schedulability check, a reschedule of all threads is called for to ensure the one with
the earliest deadline is being run. When the currently executing thread finishes execution, the routine is called again to determine the next thread eligible for execution.

The process starts by initialising two placeholders for the earliest deadline and the corresponding thread (lines 1 and 2). For each thread in the thread pool, its assigned task is obtained. Threads without tasks assigned and threads that are not marked as Active are filtered out from the process (lines 4 to 6). The status of the RTTask instance is checked to make sure the thread is marked as Validated i.e. QoS parameters are read and set. The other threads are filtered out of the process. If it is the first thread of the pool, the placeholders are set to the thread and its deadline (lines 10 to 12). The deadline of the current thread is compared with the earliest deadline found so far (line 14). If the current thread has an earlier deadline, the earliest deadline and thread placeholders are set to the current thread and its deadline (lines 15 to 16). The priority of the compared thread is set to Mid level (line 18). The thread indicated by the placeholder is set to the highest priority at the end (line 23). Prior to raising the priority of the thread with the earliest deadline, the current thread of execution is temporarily raised to the highest priority in the system to avoid any processor sharing. If the current thread has the earliest deadline, it is automatically brought down to a High priority level. The RTSJ scheduler with the support of the Operating System, guarantees the thread with the highest priority gets the processor to run.

Figures 17 and 18 illustrates the sequence of events that take place in the modified Axis2 SOAP Engine. Figure 17 contains the events that take place when a request arrives at the system, when it is being evaluated for acceptance and then scheduled. Whereas, Figure 18 contains the events taking place after a request is accepted for execution. Rather than classes, the components at a high level are used in the diagram. The events contained in the diagram are only a summary of the actual method calls that take place inside each class and may not map directly to a method in an actual implementation class.

6 Experimental Evaluation

In the solution described in the previous sections, the attempt was to achieve predictability in the execution of Web Services. The devised solution makes use of a deadline based scheduling algorithm where tasks are accepted by the system only of their deadlines can be met. Acceptance is decided using a schedulability check which validates a newly arrived request’s schedulability together with requests already accepted by the system. Hence, a request would have one of the following three outcomes.

1. Request accepted and executed within the deadline requested.
2. Request accepted and executed with a deadline miss.
3. Request Rejected

In this context, the success of the solution is primarily dependant on two factors. The number of requests accepted by the system and the number of requests meeting their deadlines. The goal is to achieve a higher number of tasks being accepted and to ensure the majority of them to be meeting their deadline requirement. The Task Size distribution, The execution time to deadline ration and the arrival rate will have an effect on the This section presents a summary of the experiments and the findings.
6.1 Challenges in evaluating the solution

Benchmarking candidate

This research provides a unique set of challenges in evaluating the solution experimentally. There have been no attempts to approach the predictability of web service execution in a
deadline oriented manner. Some of the related work discussed in section 4, indirectly relates to service execution. However, they fail to be candidates for benchmarking due to not being a solution directly relating to service execution and/or for not considering the actual service execution as a quality of service property, rather the message processing that place prior or post service execution. Hence, none could be used as benchmark references to evaluate the performance. Therefore, the solution is compared with the unmodified version of Axis2. Since Axis2 by default works in a best effort manner, there would be no resultant task rejections. However, the number of tasks meeting their deadlines is used as the metric to compare the performance.

Test Data Set

Due to the unavailability of a proper benchmarking candidates, there is no standard data set that can be used for evaluation purposes. Therefore, deciding the proper dataset for evaluation, poses a challenge. The three factors that could have an effect on the metrics were identified as Task Size Distribution, Execution Time to Deadline ratio and the Arrival Rate of tasks at the system. There is no indication of previous work that models web service work loads. As a result, it was decided to custom generate the requests for the experiment.
Priority inheritance in execution

The predictability of a real-time system largely depends on ensuring the priority model used for threads is ensured in a system wide manner. For instance, the thread with the highest priority would always get the processor for execution is an accepted norm. It can only be preempted at the end of the execution or by a thread with a higher priority is a fact. However, in a production scenario, there may be instances where a higher priority thread waits on a resource which is being consumed or held by a lower priority thread. The thread holding the resource would not be able to continue execution due to its lower priority. In such a scenario, a deadlock arises and it should be prevented. Real-time Java uses a priority inheritance algorithm to resolve such a situation. The lower priority thread momentarily ‘inherits’ the priority from the higher priority thread in order to finish execution. After which, the high priority thread would be able to use the resource. Due to this constraint, when certain data needs to be extracted from the server or the clients, it needs to be done in a way preventing priority inheritance.

6.2 Experiment Setup

The implemented solution was tested for various Task Size, Deadline and Arrival Rate combinations. As the solution involves an actual implementation of the algorithm, it should be tested in realistic conditions as close as possible to a production scenario. Hence, the following hardware and software setup illustrated in Figure 19 was used.

![Figure 19: Hardware and Software Setup](image)

The Real-time Apache Axis2 implementation was hosted on a server with a configuration of 2 AMD Duron Processors running at 1.7 GHz speed with 4 Gigabytes of RAM. Sun Solaris version 10 update 5 was used as the Real-time server Operating System, while Apache web server, Real-time Java version 2.1 was used as the primary software needed for the experiment. Together with the Real-time enabled version, the unmodified version was also hosted, for the benchmarking runs. Five client machines are used to send 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 are done only on the server side, it was decided to make use of a Real-time operating system and Real-time Java on client machines 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 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.

The requests are generated for a specific web service hosted on the server. The test web
service used accepts one integer value as an input parameter. The service counts the number
of prime numbers from 1 to the given number. The number of primes calculated is returned
back to the client as the result.

Algorithm 4: Counts the number of primes until a given number

| input | N: Number until prime numbers should be counted |
| output | Number of primes from 2 to N |

1. count ← 0;
2. foreach i ∈ {n: integer; 2 ≤ n ≤ N} do
3.   if isPrime (i) then
4.     count ← count + 1;
5.   end
6. end
7. return count;

Algorithm 5: isPrime: Function to check whether a given number is a Prime number

| input | N: Number to be checked as a Prime |
| output | true: if N is a prime number; false: if it is not a prime |

1. if N ≤ 2 then
2.   return false;
3. end
4. if N ∈ {even numbers} then
5.   return false;
6. end
7. foreach i ∈ {n: odd number; 3 ≤ n ≤ N} do
8.   if N is divisible by i then
9.     return false;
10. end
11. end
12. return true;

Algorithms 4 and 5 formulates the body of the web service sans the annotations that
indicates it is to be exposed as a web service. Algorithm 4 has a time complexity of $O(n)$
while algorithm 5 has a time complexity of $O(log n)$. As a result the overall time complexity
of the web method results in $O(n)$.

An experiment is setup by evenly dividing the total task size range amongst the 5 clients
used. The even distribution is not mandatory although it is done with a view to balance
the workload on a client. The controller and clients communicates with each other using a
very simple protocol. When the controller runs, it broadcasts a Universal Datagram Protocol
(UDP) request for any clients on the network. Each client running on the network replies back with an identifier and the task size range it is configured to support. The controller stores these details along with the ip address of each client in a hash table and references it to match the client that a particular request should be channelled to. When the controller starts issuing requests, it sends a self contained request to the selected client, including the task size, execution time and the deadline. The execution times is probabilistically calculated using the execution times from the profiling runs. The clients upon receiving the requests, spawns threads for each request to be converted in to a web service call to the server. The client will issue the web service request and wait for either the result or a negative acknowledgement in the form of an exception from the server. The total time for each request is measured on both the server and client ends.

6.3 Request Generation

The implementation needs to be tested with a set of requests that resembling a live production scenario as close as possible. The number of tasks accepted by the schedulability check and the number of tasks meeting their deadlines are measured on the server side. The Task Size, Deadline and the Arrival Rate needs to be varied as these properties would have an effect on the metrics. As there have been no previous work on modelling web service requests, it was decided to try out various combinations of these properties.

The task size of a web service is largely governed by the input parameters provided for execution. With the web service being used for the experiment runs, a task size range of 1 to 5000000 was considered. The task sizes are generated within this range using Uniform, Exponential and Pareto probability Distributions. These distributions are picked for A Uniform distribution gives the case of having an almost equal mixture of the various task sizes in the range. The exponential distribution tends to be more biased towards smaller task sizes than large sizes and the pareto distribution, being a ‘heavy-tailed’ distribution would have a large number of small sized tasks with a few very large tasks. Exponential and Pareto distributions are continuous probability distributions. Since the need is for the generation of task sizes within a given range of values, these distributions need to be of a bounded form. Mathematically, it is possible to derive a bounded form from a probability distribution by dividing the Probability Density Function by the bounded area of the distribution. Therefore, the two distributions used become Bounded Exponential and Bounded Pareto. Uniform distribution is inherently a bounded distribution.

In generating values from a distribution the inverse form of the probability distribution is used where the probability is given as the input and the smallest possible value for the given probability is obtained from the distribution. This method is used in generating all three properties mentioned above. The same probability of the generated task size is used in generating an execution time from the distribution bounded by the values obtained from the profiling run done offline. Furthermore, the deadline for a given request is generated by factoring the generated execution time by a value uniformly drawn out from a range.

7 Experiments and Results

The first set of experiments done was to measure the performance of the implemented algorithm in terms of the number of tasks qualifying for execution from the schedulability check and the number of tasks meeting their deadlines. The experiment is setup to have a task
size range between 1 to \(5 \times 10^6\) and the sizes are equally divided between the 5 clients. The experiment is done using all 3 probability distributions and for different Arrival Rate ranges which are uniformly distributed. The deadline factors used for multiplication is set to values between 1.5 and 10. For instance the execution time for the request will be multiplied by a value between 1.5 and 10 drawn out uniformly from the distribution. Each set of parameter are run against the Real-time implementation of Axis2 and the unmodified version of Axis2. The thread pools in both cases are set to the default number of 100 minimum and a maximum of 150. For each experiment, a total of 2500 requests are generated by the controller and disseminated among the clients. A summary of the experiment parameters used is shown in Table 3.

### 7.1 Schedulability Check - Acceptance Rate for Execution

Figure 20 illustrates the results for all three request size distributions for an arrival rate of between 0.25 - 5 seconds, run on the Axis2 with the Real-time enhancements. The pie charts show the task acceptance results as a ratio between the number of accepted and rejected tasks. The bar charts show a breakdown of the results according to request size groups mapped to each of the client machines.

The Uniform distribution creates a request mix of all sizes in the range. This is clearly visible in the bar chart with the breakdown of requests. This mix of request sizes leads to the lower acceptance rate of 41.8% compared to the other cases. The bounded exponential run with \(\lambda = 10^{-6}\) has more small sized requests and comparatively less medium and large sized requests. It results in an overall acceptance rate of 99% with only 1% of the requests rejected, being unable to meet with their deadline requirements. The remaining bounded exponential run with \(\lambda = 10^{-5}\) and the bounded pareto run with \(\alpha = 0.5\) both have only smaller size requests in the mix. With smaller deadline and execution time requirements, more of them can be accommodated in a given period of time. Hence, they result in all requests being accepted. The bounded pareto run with \(\alpha = 0.05\) has a better mix of tasks than the other bounded

<table>
<thead>
<tr>
<th>Task Size &amp; Execution Time Distribution</th>
<th>Task Size Range</th>
<th>Execution Time (milliseconds)</th>
<th>Deadline (Uniform)</th>
<th>Arrival Rate (Uniform)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 5 sec^{-1}</td>
</tr>
<tr>
<td></td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 10 sec^{-1}</td>
</tr>
<tr>
<td>Bounded Exponential (\lambda = 1 \times 10^{-6})</td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 2 sec^{-1}</td>
</tr>
<tr>
<td></td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 5 sec^{-1}</td>
</tr>
<tr>
<td></td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 10 sec^{-1}</td>
</tr>
<tr>
<td>Bounded Exponential (\lambda = 1 \times 10^{-5})</td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 2 sec^{-1}</td>
</tr>
<tr>
<td></td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 5 sec^{-1}</td>
</tr>
<tr>
<td></td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 10 sec^{-1}</td>
</tr>
<tr>
<td>Bounded Pareto (\alpha = 0.5)</td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 2 sec^{-1}</td>
</tr>
<tr>
<td></td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 5 sec^{-1}</td>
</tr>
<tr>
<td></td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 10 sec^{-1}</td>
</tr>
<tr>
<td>Bounded Pareto (\alpha = 0.05)</td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 2 sec^{-1}</td>
</tr>
<tr>
<td></td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 5 sec^{-1}</td>
</tr>
<tr>
<td></td>
<td>1 - 5 \times 10^6</td>
<td>8 - 24739</td>
<td>1.5 - 10</td>
<td>0.25 - 10 sec^{-1}</td>
</tr>
</tbody>
</table>

Table 3: Properties of Requests
However, the runs with some amount of medium and large sized requests has a drop in the acceptance rate compared to the exponential runs. However, the system still accepts 100% in both cases.

The uniformly distributed tasks the runs involved a larger arrival rate range. The first 5 sets of graphs in Figure 21 involves computation time. A larger task takes the place of several or many smaller sized requests in acceptance between more than 40% of the tasks sent to it. If the requests sizes take an exponential or a pareto type distribution, a high task acceptance rate can be expected. Another observation is, due to an almost equal mix of request sizes, the uniform distribution has less smaller request sizes accepted compared to the number of large request sizes accepted. This is due to accommodating deadline and execution time requirement of large tasks taking more computation time. A larger task takes the place of several or many smaller sized requests in terms of execution time.

7.2 Impact of Request Arrival Rate

The set of requests used, had a uniform arrival rate distribution between 0.25 to 5 seconds. To understand the effects that the arrival rate has on the task acceptance rate, a subset of the runs involved a larger arrival rate range. The first 5 sets of graphs in Figure 21 involves a larger arrival rate range of 0.25 to 10 seconds.

With the larger arrival rate range being used, requests arrive at the server less rapidly compared to the previous set of runs. The previous two cases having 100% acceptance rate remains unchanged as expected with the second set of runs. The uniformly distributed tasks achieve an 81.2% acceptance rate with the larger arrival rate range. The previous two runs having only a very few tasks rejected, achieves a 100% acceptance rate with the expanded arrival rate range. Furthermore, it can be clearly noticed that the number of smaller requests being accepted has increased compared to the number of large requests. This is due to the lower arrival rates achieved by the second run and as a result the amount of work completed by the system between the arrival of requests is higher than the previous case.

The next five sets of graphs contain results of runs with higher arrival rates. An arrival rate range of 0.25 - 2 seconds were used on the same task size ranges for the exponential and pareto distributions. As can be seen from the results, the request sets with only smaller task sizes, such as the exponential run with $\lambda = 10^{-5}$ and pareto run with $\alpha = 0.5$, does not have any impact from the higher arrival rate. The system still accepts 100% in both cases. However, the runs with some amount of medium and large sized requests has a drop in the

47
acceptance rate. Pareto run with $\alpha = 0.05$ has a drop of 0.5% compared to the first set of results, whereas the exponential run with $\lambda = 10^{-6}$ has a far more significant drop of 38.1%.

With the above observations, it can be clearly stated that the arrival rate of requests has an impact on the number of requests being accepted for execution by the real-time implementation of Axis2. High arrival rates result in a lower acceptance rate, whereas lower arrival rates would lead to higher acceptance rates. For the entire set of experiments only a Uniform distribution of arrival rates was used. Although it is possible to use distributions such as bounded exponential or pareto, even with this results it is clearly visible the possible outcomes of such scenarios. Using either of those distributions will result a higher concentration of high arrival rates comparatively in any given range. This would yield only to lower acceptance rates.

7.3 Timeliness of execution

The effectiveness of the solution not only relies on the number of requests accepted by the system but more importantly whether the accepted requests would meet their deadline requirements. The next set of graphs illustrate these statistics for each of the experiment runs visited earlier. Furthermore, each of the experiment scenarios were run against the unmodified version of Axis2. The results presented in Table 4 compares the performances of the two systems side-by-side.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Arrival Rate($sec^{-1}$)</th>
<th>Deadlines Met</th>
<th>Deadlines Missed</th>
<th>Deadlines Met</th>
<th>Deadlines Missed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>0.25 - 5</td>
<td>96.6%</td>
<td>19</td>
<td>3.4%</td>
<td>1043</td>
</tr>
<tr>
<td></td>
<td>0.25 - 10</td>
<td>83.6%</td>
<td>331</td>
<td>16.4%</td>
<td>2024</td>
</tr>
<tr>
<td>Bounded Exponential $\lambda = 10^{-6}$</td>
<td>0.25 - 2</td>
<td>42.6%</td>
<td>678</td>
<td>57.4%</td>
<td>1560</td>
</tr>
<tr>
<td></td>
<td>0.25 - 5</td>
<td>100%</td>
<td>2475</td>
<td>0%</td>
<td>2478</td>
</tr>
<tr>
<td></td>
<td>0.25 - 10</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Exponential $\lambda = 10^{-5}$</td>
<td>0.25 - 2</td>
<td>-</td>
<td>2491</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.25 - 5</td>
<td>-</td>
<td>2484</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.25 - 10</td>
<td>-</td>
<td>2495</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td>Bounded Pareto $\alpha = 0.5$</td>
<td>0.25 - 2</td>
<td>-</td>
<td>2496</td>
<td>100%</td>
<td>2489</td>
</tr>
<tr>
<td></td>
<td>0.25 - 5</td>
<td>-</td>
<td>2500</td>
<td>100%</td>
<td>2497</td>
</tr>
<tr>
<td></td>
<td>0.25 - 10</td>
<td>-</td>
<td>2493</td>
<td>100%</td>
<td>2493</td>
</tr>
<tr>
<td>Bounded Pareto $\alpha = 0.05$</td>
<td>0.25 - 2</td>
<td>-</td>
<td>2481</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.25 - 5</td>
<td>-</td>
<td>2494</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.25 - 10</td>
<td>-</td>
<td>2490</td>
<td>100%</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4: Comparison of Unmodified Axis2 and Real-time Axis2 : Deadlines Missed/Achieved
Figure 21: Experiment Run: $1 - 5 \times 10^6$ Task Size Range; Varying Arrival Rate

Table 4 contains deadlines missed and achieved by the two systems for all runs discussed previously. Furthermore, Figure 22 shows the execution times of the tasks in milliseconds, against their task sizes for both versions of Axis2. Note that Axis2 runs in a best effort scenario where it will try to execute all requests it receives in parallel. However, at times this leads to a build-up of threads on the server where all of them are sharing the processor. This leads to a higher resource consumption and the system becomes unresponsive at times. Moreover,
this phenomenon also results in a thread build-up on the client side where unresponsiveness and lack of memory could lead to all requests not being serviced. In such scenarios, every accepted task, considered for the deadline achievement metric, might not have successfully finished execution by the time an out of memory exception occurs. Therefore the total number of requests serviced by the SOAP engine would not calculate up to the full 2500.

The uniform distributions contain a greater mix of the task sizes than the other two distributions. As a result their rejection rates were comparatively higher. When unmodified Axis2 is faced with the same set of requests, it tries to execute all requests to a completion. With a 0.25 - 0.5sec uniform runs, the unmodified version of Axis2 misses 541 of the deadlines. In the second run with a lower arrival rate range it misses 169 of the deadlines. The reason for the low number of deadline misses with the higher arrival rate is due to most of the requests timing out or crashing due to being out of memory as a result of the thread build up. However, if the number of deadline misses is considered as a percentage, the case with the higher arrival rate misses 96.6% of the deadlines. In the scenario with the lower arrival rate, 83.6% of the deadlines are missed by unmodified Axis2. In both scenarios, the real-time implementation of Axis2 meets the deadlines 100%. If the execution times for request sizes are considered for both cases, it can clearly be seen from the respective graphs in Figure 22 that the real-time version of Axis2 outperforms the unmodified version by a large factor. Moreover, it can clearly be seen that with the unmodified version of Axis2, the execution times of the requests vary within a large range. This is due to the impact other requests may have on a given request when all of them are competing of the processor.

If the bounded exponential scenarios are considered, there is generally a higher number of small sized requests compared to the medium and large sized requests. Both implementations of Axis2 achieves all the deadlines. While, Real-time Axis2 benefits from the algorithm that ensures the deadlines are met, the unmodified version achieves deadlines due to the average execution time requirement being low. With requests finishing execution in a short period of time, a worker thread is able to finish execution in less number of pre-emptions. This prevents the build up of requests competing for processor time and all requests finish execution within their deadline requirement as a result. Although the set of runs with $\lambda = 10^{-6}$ has a number of medium and large requests, it is a low number compared to the number of small sized requests in the system. Therefore, those requests does not have a significant effect on the execution of the smaller requests in the system.
In the bounded pareto runs, the number of small sized requests in the system is far greater than the exponential case. Moreover, the sizes of the small requests tend to be closer to the lower range of the task sizes. This results in a very high concentration of very small requests at the system. Unmodified Axis2 benefits from this and as can be seen from the table, it achieves all the deadlines requested by the tasks. With Real-time Axis2, there’s a 100% acceptance rate. However, with the set of runs having \( \alpha = 0.5 \), Real-time Axis2 misses the deadlines of a very few tasks accepted. The schedulability check and the additional processing required by the Real-time scheduling algorithm, can be an overhead for requests with very small sizes. This is due to additional processing time required by these being higher than that of the actual execution time requirement of a very small task. However, the number of deadlines misses in both cases is less than 1%.

With the execution time comparisons in the bounded exponential and pareto runs, it is visible that unmodified Axis2 results in lower execution times than that of Real-time Axis2. Unmodified Axis2 is optimized for task execution in the best effort possible. Thus, requests are executed in parallel with the only limitation being the number of worker threads available in the thread pool. Although threads assigned with a request would be executing in a processor sharing manner, due to the small size of the requests in these runs, it is possible for the threads to finish executing these requests in very few pre-emption cycles. In the case of Real-time Axis2, the requests are executed in a sequential and a pre-emptive manner. Furthermore, the schedulability check and the real-time scheduling algorithm adds an additional overhead in to request processing. Although the overhead is very low or negligible for most cases, it could be significant for very small requests. Hence, it can be seen that Real-time Axis2 results in marginally higher execution times for such tasks. However, in most scenarios predominantly smaller requests Real-time Axis2 results in almost equal execution times.

Considering timeliness of execution, it would be fair to conclude that Real-time Axis2 achieves the goal of maximizing request deadline achievement. In most cases, 100% of the requests achieve their deadlines. The lowest reported percentage is 99.7%. Whenever there is a good mix of requests, it can be concluded that Real-time Axis2 performs better than Unmodified Axis2 in meeting request deadlines. In most cases, Real-time Axis2 is able to maintain useful acceptance rates of more than 62%. Moreover, with such a request composition, Real-time Axis2 results in execution times that are better than unmodified Axis2 by very large factors. When the requests are predominantly small, both implementations achieve deadlines of almost all the requests. Unmodified Axis2 does marginally better in a few cases due to the schedulability check and the real-time algorithm resulting in a slight overhead for considerably small tasks. However, this difference is negligible. In terms of execution times achieved in scenarios with predominantly small tasks, both systems perform equally where most of the resultant execution times are equal. In some of the bounded exponential and pareto runs, Real-time Axis2 execution times have resulted in higher than normal execution times for certain task sizes. These are deviations intended behaviour by the real-time scheduler. Furthermore, these are magnified on the graph by the fact that those graphs have a lesser execution time range on the y axis compared to the uniform scenarios, where the deviations exists although not being clearly visible due to the large range of execution times on the y axis. The primary goal of the real-time implementation is not to achieve better execution times than the unmodified version, but to ensure that a task finishes within a given deadline. While better execution times than unmodified Axis2 could be a by-product of the real-time scheduling, certain requests can experience longer execution times but remaining within their expected deadlines, in order to achieve the deadline of one or more other requests.
7.4 Task Rejection

In certain experiment scenarios discussed earlier, the Real-time Axis2 implementation resulted in a certain levels of request rejection. In a scenario where all requests are considered to be real-time requests, this is essential to ensure that the system could meet the deadline requirements of all requests accepted. From the experiment results discussed in the previous sub-sections, it is clearly visible that implementation performs reasonably well at it.

Although the acceptance or rejection of a request is mandated by the schedulability check, a rejection should be justified by the execution time requirement of the request and the availability of processor time depending on already accepted requests. For instance, the real-time scheduling algorithm must ensure the processor is used as much as possible. Hypothetically, it is reasonable to assume that the processor must be busy 100% of the time. However, in reality this may not be the case.

The predictability of execution of a process is dependant on various factors. Primarily the operating system must support it. If process spawns multiple threads, the predictability of execution may be relevant at a fine grain level of a thread. It could get more complicated by how the development platform and the language handles concurrency constructs and parallelism. For instance, if the priority of a thread is set to the highest possible level, the selected level would trivially be the highest available by the development platform. This differs from the priority levels used at the operating system level although a mapping between these two sets would exist. Even if the highest priority level on a development platform is mapped to the highest possible on the OS, there can be other factors that have an impact on the execution of the process. One such example is Priority inheritance or Priority inversion that could take place. With such concepts in place, practically a process would not have the CPU for it’s lifetime even if the priority on it is set to the highest available at the OS level. However, averaging the CPU utilization for a time period, the usage pattern of the process should exhibit a higher usage comparatively to the other processes in the system. With this intention, the rejection of tasks is validated against the CPU utilization of the process in this discussion.

Figure 23 illustrates the processor utilization measured per second for specifically the Real-time Axis2 process. The measurement is in fact done for the Real-time java runtime environment process which in-turn runs only Real-time Axis2. The uniform distribution runs resulted in comparatively high request rejection rate. However, in both runs it manages to achieve 100% of the task deadlines. If the processor utilization for Axis2 is considered, it is clearly visible that it remains between 80 - 95% for most part of the experiment. This is a clear indication that the processor is being used for most part of the time by Axis2. Even the uniform run with the lower arrival rate displays a very high utilization times by Real-time Axis2, justifying the resultant rejection ratio of 18.8%.

The bounded exponential run with the higher arrival rate results in a 0.1% of the requests. From the utilization graph it is clearly visible that the processor utilization was still at a very high level when the rejections happened. Therefore, the rejection is easily justified. The remaining bounded exponential runs result in 100% acceptance rates. From the graphs it is clearly visible that the utilization rate has also dropped together with the arrival rates. In the 0.25 - 5sec run, the utilization averages around 60%, whereas in the run with 0.25 - 10sec arrival rate, it falls down to even lower than 5%. This is due to the bounded exponential run with $\lambda = 10^{-5}$ being predominantly a small task mix. Due to their low execution time requirements, the load on the server is less. Hence, they all get accepted and executed.
within their deadline requirements. This phenomenon is prevalent in the subsequent bounded exponential runs for the same reason.

The three bounded pareto runs with $\alpha = 0.5$ results in a very low utilization rate through the experiment. The pareto runs result in task mixes with very small sized requests being predominant and their average sizes being very close to the lower end of the smallest range. Therefore the low utilization rates come as no surprise and it is due to the same reason discussed above for bounded exponential scenarios. The three bounded pareto runs with $\alpha = 0.05$ has a better task mix compared to the other, representing a better heavy-tailed distribution. The medium and large tasks in this mix, although being a few in numbers, makes the utilization significantly higher than the other pareto runs. The run with the
highest arrival rate results in an average around 35% of utilization and as the arrival rates become lower, the utilization also decreases as expected to an average of around 20% for 0.25 - 5sec and to a low 15% with the 0.25 - 10sec run. Recall that the the pareto runs with $\alpha = 0.5$ and higher arrival rates result in a very small amount of tasks being rejected. As explained in the previous subsection, this is due to the overhead of the schedulability check and the real-time scheduling algorithm compared to the actual processing involved with the requests. The utilization of the the processor has very less or no bearing on this as a result.

In the runs where there was rejection of tasks, it is clearly visible from Figure 23 that Real-time Axis2 has a very high utilization of the process during the experiment runs. Although it is reasonable to assume that the processor should be utilized 100% of the time by a process in such scenarios, practically it may not be the case for various reasons described in the beginning of this section. With Real-time Axis2 as the scheduling happens at thread level in the implementation, There is no direct intervention that could be created on the process level of the Operating System. However, with a Real-time Operating system being used and a development platform that supports Real-time systems, it is clearly visible that very high rates of processor utilization can be achieved. Moreover, whenever tasks are rejected by the schedulability check, it could be concluded that the decision is accurate as the resultant workload will achieve very high processor utilization rates.

### 7.5 Discussion

Depending on the mixture of requests in a given period of time, the real-time algorithm accepts between more than 40% of the tasks sent to it. If the requests sizes take an exponential or a pareto type distribution, a high task acceptance rate can be expected. Another observation is, due to an almost equal mix of request sizes, the uniform distribution has less smaller request sizes accepted compared to the number of large request sizes accepted. This is due to accommodating deadline and execution time requirement of large tasks taking more computation time. A larger task takes the place of several or many smaller sized requests in terms of execution time.

With the above observations, it can be clearly stated that the arrival rate of requests has an impact on the number of requests being accepted for execution by the real-time implementation of Axis2. High arrival rates result in a lower acceptance rate, whereas lower arrival rates would lead to higher acceptance rates. For the entire set of experiments only a Uniform distribution of arrival rates was used. Although it is possible to use distributions such as bounded exponential or pareto, even with this results it is clearly visible the possible outcomes of such scenarios. Using either of those distributions will result a higher concentration of high arrival rates comparatively in any given range. This would yield only to lower acceptance rates.

Considering timeliness of execution, it would be fair to conclude that Real-time Axis2 achieves the goal of maximizing request deadline achievement. In most cases, 100% of the requests achieve their deadlines. The lowest reported percentage is 99.7%. Whenever there is a good mix of requests, it can be concluded that Real-time Axis2 performs better than Unmodified Axis2 in meeting request deadlines. In most cases, Real-time Axis2 is able to maintain useful acceptance rates of more than 62%. Moreover, with such a request composition, Real-time Axis2 results in execution times that are better than unmodified Axis2 by very large factors. When the requests are predominantly small, both implementations achieve deadlines of almost all the requests. Unmodified Axis2 does marginally better in a few cases.
due to the schedulability check and the real-time algorithm resulting in a slight overhead for considerably small tasks. However, this difference is negligible. In terms of execution times achieved in scenarios with predominantly small tasks, both systems perform equally where most of the resultant execution times are equal. In some of the bounded exponential and pareto runs, Real-time Axis2 execution times have resulted in higher than normal execution times for certain task sizes. These are deviations intended by the real-time scheduler. Furthermore, these are magnified on the graph by the fact that those graphs have a lesser execution time range on the y axis compared to the uniform scenarios, where the deviations exist although not being clearly visible due to the large range of execution times on the y axis. The primary goal of the real-time implementation is not to achieve better execution times than the unmodified version, but to ensure that a task finishes within a given deadline. While better execution times than unmodified Axis2 could be a by-product of the real-time scheduling, certain requests can experience longer execution times but remaining within their expected deadlines, in order to achieve the deadline of one or more other requests.

In the runs where there was rejection of tasks, it is clearly visible from Figure 23 that Real-time Axis2 has a very high utilization of the process during the experiment runs. Although it is reasonable to assume that the processor should be utilized 100% of the time by a process in such scenarios, practically it may not be the case for various reasons described in the beginning of this section. With Real-time Axis2 as the scheduling happens at thread level in the implementation, there is no direct intervention that could be created on the process level of the Operating System. However, with a Real-time Operating system being used and a development platform that supports Real-time systems, it is clearly visible that very high rates of processor utilization can be achieved. Moreover, whenever tasks are rejected by the schedulability check, it could be concluded that the decision is accurate as the resultant workload will achieve very high processor utilization rates.

8 Conclusion

An approach for enhancing the performance of SOAP based web services by means of service differentiation and predictability was presented in this report. The motivation for this research was the lack of predictability in SOAP based web services and as a result the inability for applications with Real-time requirements to use them as middleware. Although there have been attempts at differentiating services on a SOAP engine, none of the methods had a clear guarantee on meeting specific QoS requirements and service times. The dynamic nature of web services and their usage poses a complex scenario where properties of the next request arriving at the server is unknown. A model and an algorithm was proposed to achieve service differentiation and predictability where incoming requests would indicate a QoS parameter in the form of a deadline that should be considered in scheduling them. The model proposes a schedulability check which ensures that a given task could be scheduled to meet the deadline requirement it is tagged with without compromising the deadlines of the tasks that have already been accepted. In the case of it being unable to schedule it the task is rejected. This rejection ensures that all tasks accepted will meet their deadlines. Empirically it was proven that the task rejection happens in an accurate manner and the resultant requests accepted contribute to a very high processor utilization. The proposed model and algorithm was implemented in a real life widely used SOAP engine.

The implementation was described in detail and it’s performance was benchmarked against
the unmodified version of the SOAP engine. Both versions were exposed to request sets created to represent uniform, exponential and pareto type probability distributions. Empirical results show that the unmodified version of the SOAP engine results in very high rate of deadline misses when faced with a mixture of requests. It performs well against requests that are small in size and are more of exponential or pareto in nature. The Real-time implementation performs well against all types of requests, resulting in on average 99.9% of the deadlines being achieved all the time. Furthermore, it betters the unmodified version of Axis2 in resultant execution times by very large factors.

References


[26] J.R. Jackson and CALIFORNIA UNIV LOS ANGELES NUMERICAL ANALYSIS RE-


ws.dev.java.net/.

us/netframework/aa663324.aspx.


[32] R.A. Van Engelen and K.A. Gallivan. The gSOAP toolkit for web services and peer-to-


9 Appendix - SOAP Engine Architecture

A SOAP Engine’s functionality goes beyond mere parameter marshalling. Web services are hosted on a server, using a SOAP engine. Therefore 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, some are based on extensibility etc. This makes the architectures of SOAP engines quite different to each other. However, it is possible to identify parts of discrete functionality to do with the fundamental task of SOAP message processing, as illustrated in Figure 24.

The SOAP engine uses a pool of worker threads to handle all types of SOAP processing requests. Once a task is assigned to a thread, the execution through each module is managed by the worker thread. This happens by default, in a best effort mechanism in every SOAP engine. The total number of tasks being handled concurrently, depends on the number of active threads in the thread pool according to it’s implementation. Hence, there would be resource contention amongst the active worker threads. This results in the execution time of a thread being dependant on the number of active threads currently serving requests, naturally taking longer time to complete when many requests are being serviced.

SOAP engines can be used both at the client and server ends. Although the processing that take 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.

Transport Module

The Transport module acts as the gateway into the SOAP engine. Its primary functionality is to listen to requests on the network. Upon receiving a request, it 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, SOAP action etc. The created 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 carry 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, achieving reliability etc. Several WS-* standards make use of the headers. Conformity to such standards and related processing happens within this module.

Serializer / Deserializer

Serialization 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 an RPC call. When a server receives a request, the de-serialization 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 call or at a server when a client request reaches the SOAP engine there.

Dispatcher

At the server end, the same SOAP engine is generally used to host more than one web service. These can be identified by their 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 message is handed over to it in de-serialized and decoded form.

Thread Pool

A SOAP engine is capable of handling multiple requests for SOAP message processing. A server would naturally contain multiple web services deployed. Therefore, receiving multiple requests for multiple services is commonality. SOAP engines use 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 of closing the connections, signifies the completion of request processed. After completing the
request, the thread is either destroyed or included back into the pool as a free thread based on the SOAP engines design.

**Client API**

Most SOAP engines provide client side functionality through a set of Application Programming Interfaces (API).