World Wide Web (2012) 15:685–744
DOI 10.1007/s11280-012-0160-x

Building Web services middleware with predictable execution times

Vidura Gamini Abhaya · Zahir Tari · Peter Bertok

Received: 16 April 2011 / Revised: 4 January 2012 /
Accepted: 7 March 2012 / Published online: 28 March 2012

Abstract Predictability of execution has seldom been considered important in the design of Web services middleware. However, with the paradigm shift brought by cloud computing and with offerings of Platforms and Infrastructure as services, execution level predictability is mandating an increased importance. Existing Web services middleware are optimised for throughput with unconditional acceptance of requests and execution in a *best-effort* manner. While achieving perceived levels of throughput, they also result in highly unpredictable execution times. This paper presents a generic set of guidelines, algorithms and software engineering techniques that enable service execution to complete within a given deadline. The proposed algorithms accept requests for execution based on their laxity and executes them to meet requested deadlines. An introduced admission control mechanism results in a large range of laxities, enabling more requests to be scheduled together by phasing out their execution. Specialised development libraries and operating systems empower them with increased control over execution. Two widely used Web services middleware products were enhanced using these techniques. The two systems are compared with their unmodified versions to measure the predictability gain achieved. Empirical evidence confirms that the enhancements made enable these systems to achieve more than 90% of the deadlines under any type of traffic, while the unmodified versions achieve less than 10% of the deadlines in high traffic conditions. Predictability of execution achieved through these techniques, would open up new

V. Gamini Abhaya (✉) · Z. Tari · P. Bertok
School of Computer Science and Information Technology,
RMIT University, Melbourne, VIC, Australia
e-mail: vidura.abhaya@rmit.edu.au

Z. Tari
e-mail: zahir.tari@rmit.edu.au

P. Bertok
e-mail: peter.bertok@rmit.edu.au

application areas such as industrial control systems, avionics, robotics and financial trading systems to the use of Web services as a middleware platform.

Keywords Web services · middleware · predictability of execution · real-time systems · earliest deadline first · scheduling

1 Introduction

"The network is the computer" is no more a mere marketing term or a futuristic idea. The Internet has evolved beyond its traditional role of being a large collection of information sources into the true form of a large collection of distributed systems. While this may have been envisioned at the inception of the Internet, its success largely rested on the software technology that enabled it. This paradigm shift was only possible with the advent of Web services technology, which is now considered to be the *de-facto* standard for communication in distributed systems [11]. Web services played the role of a change agent, evolving ways of Internet usage over the years [10]. It has enabled users around the globe to use distributed systems in many different ways, not limiting to the use of applications. Being the precursor to Software being offered as a Service (SaaS), Web services currently power service offerings such as Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). Being the pillars of cloud computing, these have enabled users to completely move away from investments in hardware infrastructure and use services for all their computing needs. Nevertheless, the true success of PaaS and IaaS offerings largely rely on the performance aspects of Web services, with an increased importance on service execution. While Quality of Service (QoS) aspects of service execution has been widely researched, few attempts have been made on achieving execution time QoS in Web services middleware. Moreover, none of them can guarantee on achieving predictable execution times in a consistent manner.

Predictability of execution is seldom considered as a design goal in developing middleware platforms. On the contrary, they are designed to achieve high levels of throughput. Similarly, Web services middleware contain many optimisations for throughput [2, 16, 22]. For instance, requests are accepted for execution unconditionally and executed in a *best-effort* manner. Multiple requests are executed in parallel using processor sharing, following the *Thread-pool* concurrency pattern [13]. Moreover, executions of all requests are treated with equal priority. Although employing such techniques yield a higher throughput, they become detrimental to the predictability of request execution. For instance, processor sharing leads to an increase in average execution time proportional to the number of requests being executed in parallel. The execution time of a request becomes highly unpredictable as it varies with the number of requests being executed concurrently at a given time, their execution times and request arrival rates. Moreover, this leads to longer and unpredictable waiting times. These reasons make Web services unsuitable for applications with stringent execution time requirements. Furthermore, the development platforms and operating systems (OS) used to build and host such middleware do not support execution level predictability. For instance, priority levels available to middleware may not orthogonally map onto OS level priorities. As a result, service execution maybe interrupted by other processes running in the system or by house keeping activities within the development platform, such as garbage collection [3].

Existing work on execution time QoS in both stand-alone and cluster based Web services middleware, can be found in literature. Some of them [25, 28] make the assumption that these middleware would guarantee promised levels of QoS and use the data to match clients to services or to aid service compositions. Others [9, 17, 26] segregate requests into multiple classes and try to achieve different levels of QoS in service execution, based on pre-defined Service Level Agreements (SLAs). These SLAs typically guarantee a probabilistic measure of execution time or a pre-defined ratio of processing among the classes. The basis of segregation may not be related to the actual performance need of the client, rather economical aspects such as preference based on payment levels of clients. Some [19] differentiate based on non-functional attributes such as security level, computational device being used and nature of client. While all of them have in common the aspect of differentiation in request execution, they are unable to achieve predictable service execution times due to a couple of reasons. Firstly, the predictability of execution depends on the guarantee that a service invocation will complete within a set period of time (or a *deadline*) repeatedly and consistently. Therein, none of them consider such a deadline as an attribute in request processing. Secondly, the software used in them are not designed to guarantee such a deadline in the middleware itself and in the supporting software, such as the development platform and the OS used. Moreover, some of them have no means of avoiding overload conditions and resource contentions that may arise with *best-effort* processing and unconditional acceptance of requests.

Techniques that facilitate such stringent execution level predictability can be found in real-time systems, where predictability of execution is considered equally important to the correctness of an operation [21]. Such systems mandate the completion of a task within a requested deadline, where even a correct result obtained with a deadline miss is considered useless. This area in computer science contains scheduling principles and techniques that facilitate the adherence to such deadlines in a consistent manner.

Research questions and aims Applications such as industrial control systems, avionics systems, medical equipment control systems, capital market trading systems and robotics mandate such stringent predictability in execution. Such that real-time requirements can greatly benefit from the inherent advantages Web services bring in, such as open protocols that would free component designers and developers from using proprietary methods of communication in such systems. Through this research we hope to open up such new application avenues to the use of Web services as a viable middleware. We aim to test the feasibility of using Web services with predictable service execution introduced through scheduling and differentiated request processing. In the process, we answer the following questions. (1) How can service execution be guaranteed practically to complete within a requested deadline? (2) Can a request be guaranteed of its deadline prior to acceptance for execution? (3) What design changes are required in current Web services middleware to achieve predictability of execution? (4) What support software does the middleware require to guarantee predictability?

Outline of the solution The success of using Web services as middleware for systems that require high levels of predictability, depends on each service invocation completing within a given deadline. Our solution for achieving predictability of service execution can be summarised functionally into three important steps. Firstly,

the requests must be explicitly scheduled to meet their deadline requirement. Such precise scheduling guarantees the most important aspect of meeting each processing deadline. Secondly, requests must be consciously selected for execution based on their laxity, to complement already accepted requests. Laxity based selection is a further guarantee on meeting the deadlines of requests being selected, without compromising the ones already accepted. Finally, the middleware must be supported by a software infrastructure (development platform and OS) that provides predictability of execution. This step ensures the predictability features implemented in the middleware are supported bottom-up by all layers of software providing the required guarantees of execution. Following these guidelines, we developed a set of algorithms and software engineering techniques that can be directly applied to enhance or build Web services middleware. These algorithms and techniques were evaluated for the predictability gain they achieve, by implementing them in two popular Web services middleware products used in stand-alone and cluster based deployments.

Contributions In our previous work [7], we presented an analytical model that uses real-time scheduling principles to calculate the schedulability of a request, on a single server. This was further extended to a cluster based model in [8], where a series of request dispatching algorithms that follow the analytical model, were presented. Both of these were theoretical attempts at solving the problem. The main contribution in this paper is a thorough analysis of the practical aspects in our solution, with an extensive discussion on challenges faced and how they were overcome. It is followed by a more comprehensive evaluation of the systems previously published.

Initially, we provide a set of guidelines to follow in either building Web services middleware or enhancing existing ones, formulated through our research in real-time systems. Following these guidelines, we developed a set of algorithms and software engineering techniques that can be directly applied to enhance or build Web services middleware. Through the use of a case study, we demonstrate how these guidelines, algorithms and techniques were used in enhancing widely used middleware products in a stand-alone and cluster based environment. Although specific products were chosen for this reference implementation, the guidelines, algorithms and techniques are generic enough to be directly applied in any Web services middleware available. The uniqueness of the techniques presented, is in the use of real-time scheduling principles (that are typically used at design time of a system), in a highly dynamic environment to make decisions at run-time. The use of the proposed techniques fundamentally work by selective acceptance of requests based on laxity, which creates a large range of laxities at the server, enabling more requests competing for overlapping windows of time to be scheduled together. The selected requests are explicitly scheduled based on their deadlines, using Earliest Deadline First (EDF) scheduling principle [21].

The suggested predictability enhancements are only applicable to Web services middleware. Herein, we reach the viable limit of what could be controlled by us for this research, as Web services can span across multiple application boundaries. For instance, the execution of a Web service using a database would go beyond the realm of the Web services middleware. The processing in the database will be part of the overall execution. Ensuring predictability of execution in applications outside the Web services middleware are research areas on their own requiring specialised attention. Therefore, they were considered out of scope for this research. However, it is strongly recommended that the enhanced Web services middleware be hosted

on dedicated server hardware and supplementary applications such as databases be hosted separately. By using compatible applications that support predictability features will ensure processing priorities are maintained across application boundaries and allow more control over the overall execution time of a Web service request. While such a combination would lead to better predictability results, it is still feasible to use enhanced Web services middleware with products that lack predictability features. The execution that happens outside the Web services middleware has to be considered as part of the overall execution time of a request. This must be considered when specifying deadlines for the Web service invocations.

The rest of this paper is organised as follows. First we present the analytical model used in the schedulability analysis algorithm as a background to the implementation. In Section 3 we present a set of guidelines that will enable Web services middleware to achieve predictable execution times. It is followed by the case study (Section 4) where generic techniques that could be used to enhance middleware products are presented together with product specific implementation details. Next, in Section 5 we present a sample scenario that explains in detail how the laxity based selective acceptance of requests and deadline based scheduling work together in our solution. Section 6 contains a comprehensive empirical evaluation of our solution. We discuss some of the related work in this area in Section 7 and provide a conclusion at the end in Section 8. Network communication aspects of Web services is considered out of scope for this research and any delays in the network are not quantified separately. Rather, it is subsumed within the execution time of a service in the empirical evaluation. The terms *request* and *task* are used interchangeably to refer to a Web service request throughout the paper.

2 Background

In this section we present the essentials of the analytical model presented in [7] used to calculate the schedulability of a Web service request, given other requests already executing in a system. The model based on real-time scheduling principles, forms the theoretical basis for the schedulability check algorithm presented in this paper. The model highlights two concepts namely *Processor demand* and *Loading factor* from the area of schedulability analysis in real-time scheduling [21]. Henceforth, we use a given task T_i, with release time of r_i, a deadline of d_i and an execution time requirement of C_i. We consider a semi-closed time interval that is left open and right closed depicted as $[a, b)$ where $\{x \in \mathbb{R} \mid a \leq x < b\}$.

The analytical model is described considering the scenario of a request arriving at a system with already accepted requests in execution. The new arrival is accepted for execution subjected to the schedulability check formed by the model. Herein the system considered executes requests using pre-emptive scheduling, where the execution of a given request could complete with several pre-emption cycles.

Definition 1 For a given request T_i having n number of pre-emptions, where the start time of each execution is s_n and the end time of each execution is e_n, Total time of the task execution E_i can be considered as,

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

Definition 2 For a given request submitted to the system, with an execution time requirement of C_i, at any given point of time the remaining execution time R_i can be considered as,

$$R_i = C_i - E_i. \tag{2}$$

When a newly submitted task arrives at the system, the schedulability check is done to ensure it could successfully be scheduled together with the tasks already in the system. The proposed schedulability check calculates the processing requirement of the new task against the tasks in the system, in a number of cycles. First, a segregation of the currently accepted tasks is done, on the basis of whether the deadline of a task 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_{new} be a newly submitted task, with a release time of r_{new} and a deadline of d_{new} and an execution time requirement of C_{new}. Let P be the set of tasks already accepted and active in the system, with their deadlines denoted as d_p.

With reference to the formal definition presented in [7], the processor demand within the duration of the newly submitted task can be defined as,

$$h_{[r_{\text{new}},d_{\text{new}})} = \sum_{r_{\text{new}} \le d_p \le d_{\text{new}}} R_p + C_{\text{new}}. \tag{3}$$

With reference to the formal definition presented in [7], the loading factor within the duration of the newly submitted task can be defined as,

$$u_{[r_{\text{new}},d_{\text{new}})} = \frac{h_{[r_{\text{new}},d_{\text{new}})}}{d_{\text{new}} - r_{\text{new}}}. \tag{4}$$

With condition (4), if the following condition is satisfied, the new task is considered schedulable together with tasks finishing on or before its deadline, with no impact on their deadlines.

$$u_{[r_{\text{new}},d_{\text{new}})} \le 1. \tag{5}$$

With the above condition (5) satisfied, the task is checked for schedulability with the tasks finishing subsequently. Unlike (3), the calculation of processor demand needs to be done for each task with deadlines after d_{new}, separately.

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

$$h_{[r_{\text{new}},d_q)} = h_{[r_{\text{new}},d_{\text{new}})} + \sum_{d_{\text{new}} \le d_i \le d_q} R_i. \tag{6}$$

The result of (3) is used as part of the equation. This represents the processor demand of all tasks finishing on or prior to d_{new} and can be treated as one big task with a release time r_{new} and a deadline of d_{new} respectively. Next, the loading factor for the same duration is calculated.

$$u_{[r_{\text{new}},d_q)} = \frac{h_{[r_{\text{new}},d_q)}}{d_q - r_{\text{new}}}. \tag{7}$$

The loading factor is also calculated per task 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 considered as schedulable.

$$u_{[r_{\text{new}},d_q)} \leq 1. \tag{8}$$

In summary, for a newly submitted task to be accepted for execution, condition (5) needs to be satisfied for tasks with deadlines on or before d_{new}, subsequently condition (8) needs to be satisfied, separately for each task with deadlines after d_{new}.

3 Guidelines

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

As illustrated in Figure 1, Web services deployed within a server are exposed through the middleware to the outside world. The middleware handles all requests (SOAP and REST) with the aid of many development platform libraries that provide message processing and network level communication. In turn, the functionality provided by the development libraries are facilitated by the underlying OS. The OS handles the execution of threads, processes and manages system level resources such as CPU time, memory, sockets for network communication, access to Input/Output devices and other peripherals. For managing resource allocations, execution of processes and threads, the OS uses system level priorities for differentiation. These priorities, can be requested by the development platform or defaulted to OS preferences. The OS decides on the precedence of execution and resource allocation based on such priorities. Therefore, any form of predictability at the upper layers of a software stack, is only achieved with the support of all underlying layers.

From our research into the design of real-time systems, we present the following guidelines that would enable Web services middleware to achieve predictability of execution. At a high level, these cover functional and software engineering aspects

Figure 1 Default software stack.

that could be used to enhance existing Web services middleware, or when they are newly developed.

G1. **Use of an operating system, development platform and libraries with predictability features.**
Predictability of execution in a server can only be achieved if such features are provided by the lower layers of software being used. Most widely used development platforms and operating systems are intended for general use, thus have no support for predictable execution. For instance, thread priority levels used in the standard and enterprise versions of the Java development platform do not directly map to the range of priorities available at the OS level [23]. As a result, the execution of a thread running at the highest priority available in Java, can be interrupted by other processes running with higher OS level priorities. Similarly, it could also be interrupted within the platform itself, by housekeeping activities such as garbage collection [3]. The use of specialised real-time development platforms and OSs ensure predictability by having features such as high precision clocks, fast context switches with minimum overhead, guaranteed priority levels, fast memory based I/O, faster responses to interrupts and priority inheritance mechanisms [20]. Figure 2 depicts such a setup with the software required in all levels.

G2. **Support deadlines for service execution and decisively schedule requests to meet them.**
The invocation of a Web service typically happens semantically equivalent to a method call of an object, where input parameters are specified and a result is returned. For the invocation to always complete within a target, Web services middleware must be specified with a time limit. Therefore, middleware supporting predictability of execution must introduce means of specifying a user perceived deadline. Subsequently, the middleware must explicitly schedule the service invocation to meet the specified deadline. However, in the event of multiple invocations having overlapping executions, the middleware must ensure that scheduling a request based on its deadline does not compromise the others with overlapping lifetimes.
Figure 3 depicts a schedule of tasks based on their deadlines. Herein, tasks are executed in the increasing order of their deadlines and completes execution in the order of T3, T4, T5, T2 and T1. On its arrival, each task has an overlapping

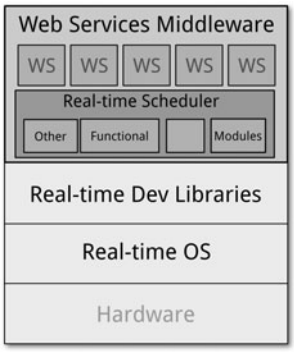

Figure 2 Required software stack.

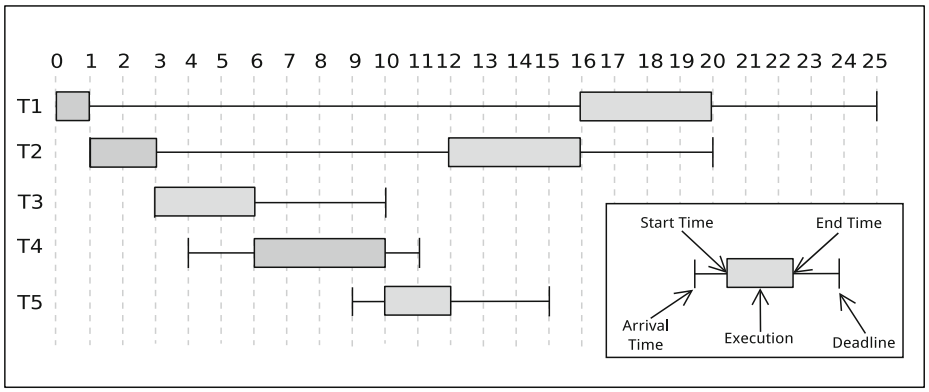

Figure 3 Deadline based task schedule.

lifespan with one or more tasks already in execution. However, tasks with earlier deadlines have been able to finish their execution within the required time limit, as a result of being explicitly scheduled on this basis. Although having arrived at the system later, tasks T3, T4 and T5 have been explicitly scheduled to complete prior to T1 and T2 by pre-empting them from execution. Subsequently, T1 and T2 also achieve their deadlines despite being executed in a staggered manner due to their longer deadlines (large laxities).

G3. **Conditionally accept requests for execution based on their laxity property.**
The lifetime of a task arriving at a system is determined by its arrival time (assuming it is ready for execution) and the perceived deadline, before which it must complete execution. Depending on arrival rates of requests, it is quite common for them to have overlapping lifespans. While scheduling these tasks with a deadline guarantee, maybe possible by delaying the execution of unfinished tasks with longer deadlines, there will be instances where staggered execution of tasks would not be possible without a deadline miss. To ensure fairness, the deadline of an already accepted task should not be compromised for a new task even though it maybe having an earlier deadline. Therefore, it is imperative that requests must be accepted for execution conditionally, ensuring deadlines of other requests are not compromised.
The laxity property of a request is an indicator on the possible delay of execution while meeting its deadline requirement. A larger laxity enables the execution of a request to be delayed safely, thereby allowing more requests to be scheduled together. Scheduling a given set of requests ensuring their deadlines, is only possible with a greater variety of laxities within them. For instance, meeting the deadlines of tasks T3, T4 and T5 in Figure 3, was only possible due to the larger laxities of T1 and T2 resulting in their delayed, phased out execution. Similarly, the shorter laxities of T3, T4 and T5 enabled them to achieve their deadlines within the lifespan of T1 and T2. Conversely, T4 may not have been able to achieve its deadline executing together with T3 and T5, if it had a smaller laxity. Therefore, requests must be consciously selected for execution, resulting in a large range of laxities at the server.

G4. **Achieve differentiated request processing at system level.**
The invocation of a Web service has many steps to be completed by different components inside the Web services middleware. Common to any such middleware, the execution of a request is typically handled by one or more worker threads (the smallest unit of execution) throughout its entire lifetime within the middleware. While the execution times at each component may vary depending on the nature of processing, the individual times are subsumed within the overall execution time of a request. Widely used Web services middleware treats all threads equal and makes no differentiation in their processing. This results in the middleware having no control over the completion time of a request.
However, achieving predictability in execution is only possible, if some differentiation in request processing is achieved within the middleware. For instance, when a new task with an earlier deadline arrives at the system in Figure 3, the execution of the current task has to be suspended and resumed at a later point of time. It must be possible for the server to suspend the execution of one task (e.g. T2 and T1) and let another start execution (e.g. T3). Therefore, at any given time the middleware must be able to control which thread is in execution and which is suspended. This fine-grain control will allow the middleware to decide on how the processing resources are consumed by the smallest units of execution. Such control will enable the middleware to avoid deadlocks and unnecessary delays on execution due to resource unavailability. A properly managed set of priorities makes it possible to achieve such fine grain control over the execution of threads.

G5. **Reduce instances of possible priority inversions.**
Contention for system resources is often encountered in task execution. Another form of delay that maybe added to the execution of a request is the possibility of a priority inversion. This refers to the scenario where a resource required by a higher priority process or a thread is held by a lower priority process or thread [21]. As depicted in Figure 4, this could take place when the lower priority thread in execution that was consuming resource X is preempted by a higher priority thread. The higher priority thread also wishes to consume resource X to complete its execution. However, this becomes impossible as the resource is currently held by the lower priority thread which has been preempted from execution resulting in a deadlock. The hold on resource X by the lower priority thread may finally be released by a time-out, if such a mechanism is used by the OS to free unreleased resources. In which case, the high priority thread maybe able to resume execution, albeit the delay incurred by the wait.
OS design often solve such problems using priority inheritance algorithms [21]. Consumption of resource X is a necessity for the higher priority thread to complete its execution. Since its execution cannot be resumed till resource X is released, the OS makes the lower priority thread 'inherit' the higher priority temporarily, to resume its execution and release the resource. Once the resource is released, the priorities are inverted back to their original state and the execution is resumed. Although this mechanism solves the problem, it adds an unwarranted delay (equal to the execution of the priority inheritance algorithm and the controlled execution of the lower priority thread) to the

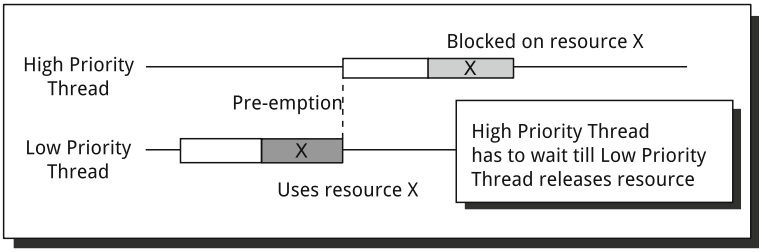

Figure 4 Priority inversion.

overall completion time of the high priority thread. Such priority inversions may happen with simple I/O operations such as writing to a file or displaying a message to the console. Moreover, they would not only create delays in the actual execution, but be responsible for unexpected results and behaviour in activities such as when debugging such applications. For instance, common debugging practices such as the use of log files and trace messages can result in unexpected priority inversions. Prevention of such phenomenon is only possible by avoiding such trivial techniques and using specialised ones instead.

Adhering to these guidelines specified, will enable Web services middleware to function with predictable execution and be successfully built accordingly. While these are valid for both SOAP and REST based Web services, the way they are implemented in various middleware products, may differ from each other. In the case study presented in Section 4, enhancements made to two widely used Web services middleware products are presented. Although these were used as examples, the enhancements are generic enough to be applied for any other middleware product available.

4 Case study

In this section we present a case study where the functionalities of two popular Web services middleware products are enhanced to achieve predictability of execution, using the guidelines presented in Section 3. Features introduced to Apache Axis2 [2] is presented as an example of how these guidelines could be used in enhancing a stand-alone Web services middleware server. The second case study is an example of how the guidelines will help in achieving predictability of execution in a cluster setup hosting Web services. We enhanced Apache Synapse [1], an Enterprise Service Bus (ESB) product to act as the dispatcher of the cluster and use the enhanced version of Axis2 as the executors hosting the Web services. Furthermore, we implemented a few specialised dispatching algorithms to distribute tasks within the cluster preserving the predictability of execution following the guidelines mentioned. The two case studies are presented as follows. For each case study, the enhancements that are generic in nature are presented first without any product specific implementation details. These can be directly implemented on any Web services middleware. Specialised changes

made to each product with specific implementation details are presented thereafter. Although product specific implementation details are presented, conceptually the techniques are still applicable to others.

4.1 Development platform and OS

Apache Axis2 and Apache Synapse have been developed using Java as the development platform. Whilst versions of Axis2 are available also in C, the fully featured Java version is preferred by developers. Moreover, Apache Synapse also uses parts of Axis2 in its core. Therefore, the Java versions of Apache Axis2 and Synapse were selected for this implementation. Java is known to be a platform that lacks predictable execution times due to features such as garbage collection [27]. Addressing this issue and conforming with the guideline G1 in Section 3, we use Java Real-time Specification (RTSJ) [23] as the supporting development environment. RTSJ introduces several features (not available in standard Java releases) that support predictability of execution required for applications with stringent time requirements. For instance, it introduces several new strictly enforced priority levels that directly map on to proper OS level counterparts. Moreover, it also contains a new real-time thread class that can be empowered with the aforementioned priorities to ensure uninterrupted execution even from the garbage collector mechanism. RTSJ also provides high precision clocks that could be used for timing in such applications upto a nanosecond accuracy.

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

4.2 Introduction of a deadline

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

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

4.3 Generic enhancements made to Axis2

Schedulability check A major change required to achieve predictability in Web services middleware, is the conditional acceptance of requests. Herein, requests must be selected based on their laxity property. Following guideline G3, every request is subjected to a schedulability check that follows Algorithm 1. Functionality of the algorithm is summarised in Figure 5. The check ensures that a request is only

Algorithm 1 Schedulability check

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

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

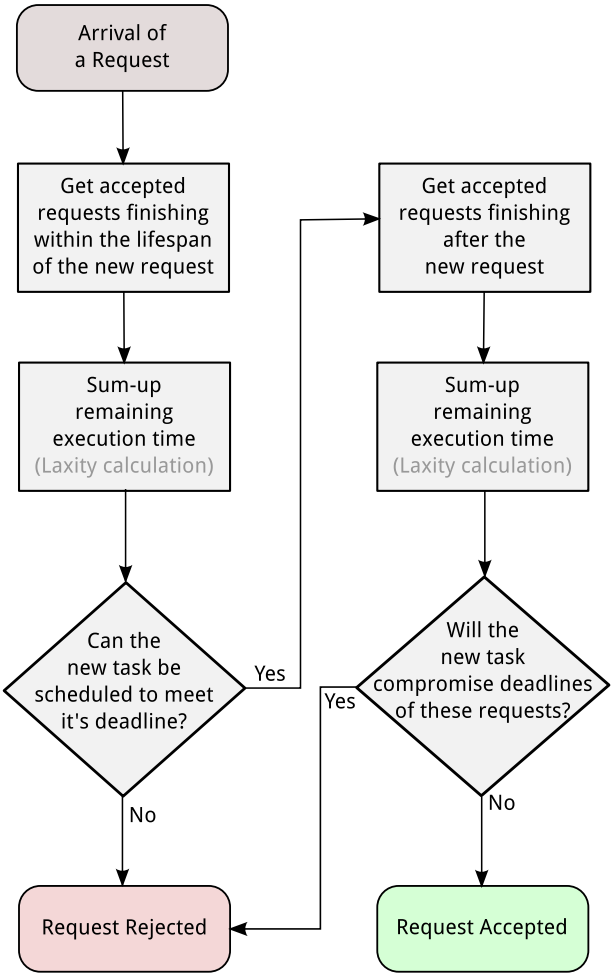

Figure 5 Schedulability check summarised.

accepted for execution if its deadline requirement could be met. Furthermore, it also ensures that deadlines of the already accepted requests will not be compromised by any controlled delays or phase-outs in execution, resulted by the acceptance of a new request. The algorithm follows the analytical model presented under background in Section 2, and differs from the algorithm presented in [7] with better execution time complexity of $O(n)$, where n is upper bound by the total number of accepted requests. The analytical model will be referred to as (*A.M.*) and the particular formulae (*Eq.*) cited where applicable, within the discussion.

On the arrival of a new request, the algorithm considers the new arrival, the queue of already accepted requests (RQ) at the server and returns whether the new request could be accepted for execution or be rejected. RQ is a priority queue with requests in the increasing order of their deadlines. The schedulability of a new request (N)

is checked in two parts. The first part checks whether the deadline requirement of the new request can be fulfilled. Remaining execution time of already accepted requests, with deadlines earlier than that of N is considered (line 7). As the execution of N would have to be delayed until their completion, the laxity of N is checked against the total remaining execution time of the others (lines 9–23). This is done by calculating the processor demand within the lifespan of N (*A.M.* (3)), where either the remaining execution times (if the request is partially executed) or the execution time requirements (if the request is yet to be executed) of the accepted requests are totalled (lines 8–12) and then execution time requirement of N is added to it (line 19–23).

Execution time history from previous invocations are stored by the real-time scheduler module introduced into Axis2. The information is stored as an average for the combination of input parameters used for the invocations. Similarly, a global average per service is also kept track of by the scheduler module. Both averages are updated at the end of each invocation. Remaining execution time of an already accepted request is calculated by deducting the time already spent in execution, from the average for its input parameters. When a set of input parameters are used on a service for the first time, the global average for the service is used in place of the missing execution time history (line 11). The first ever request handled by the system will use the requested deadline in place of the global average in that once off scenario. Next the loading factor within the lifespan of the new request (*A.M.* (4)) is calculated (line 24). The loading factor indicates the ratio between the amount of processing required within a time period and the available processing time. A loading factor of more than 100% (*A.M.* (5)) will result in N being rejected (lines 25–28). A successful loading factor leads to the second part of the check.

Second part of the schedulability check ensures the acceptance of a new request will not result in any deadline misses of already accepted requests. Hence requests with subsequent deadlines to that of N, are considered for this step (lines 14–16). As there maybe multiple such requests accepted, the acceptance of N has a domino-effect on the start of their remaining execution.

Therefore, the effect of N's execution time requirement on their individual laxities is checked incrementally (lines 30–42). The deadline ordering of requests in RQ, aids this process. Considering each request (*nextReq*) with a deadline later than that of N, the time period considered is between the start time of N and the deadline of *nextReq*. The processor demand for this period is calculated (*A.M.* (6)) by adding the already calculated value within the lifespan of N (line 29) and the remaining execution time (or global average, if yet to be executed) of *nextReq* (lines 32–36). The loading factor for the time period (*A.M.* (7)) is calculated next (line 37) and a result of more than 100% will result in N being rejected (lines 38–41). The check (*A.M.* (8)) continues on till all requests with subsequent deadlines to that of N are considered individually with successful loading factors. If all subsequent requests can be scheduled to meet their deadlines the new request is accepted for execution and inserted into the request queue (line 43). The processor demand used in each iteration of the calculation is a cumulative figure carried forward through the algorithm. This ensures that if acceptance of a request leads to a possible deadline miss in at least one of the requests already accepted, it is detected early on as possible and further processing is terminated. Moreover, the implementation also results in

an efficient execution time complexity of $O(n)$ where n is the upper bound of total requests accepted for execution.

To prevent the set of accepted requests being changed amidst the schedulability check, access to RQ is controlled using a semaphore. The schedulability check works on securing a lock on the list (line 4) and releasing it prior to exiting the schedulability check (lines 26, 39 and 44). Similarly, access to the entire check is controlled to prevent two tasks being concurrently checked for schedulability, by marking the entire algorithm as a critical section (lines 1 and 46). As a result, only a single thread would be active inside it at any given time. Requests that get accepted through the schedulability check result in a large range of laxities at the server, thereby enabling more requests to be scheduled together.

Priority model Typical Web services middleware contain no mechanisms to differentiate request processing. Therefore, all requests are executed at the same priority level. Guideline G4, mandates fine-grain differentiation in request processing, for achieving execution time predictability. Following this guideline, three priority levels are used to achieve differentiation in the functionality of the middleware. The *lowest* priority level is used for metadata exchange such as Web Service Description Language (WSDL) or Schema requests. The *highest* level of priority is used to allow the chosen request to exclusively use the CPU for execution. Similarly, the *mid-level* priority is used on remaining tasks to prevent them from using the CPU at any given time. These priority levels are used at runtime by a newly introduced real-time scheduler component, to achieve fine-grain differentiation in request processing.

Real-time scheduler and thread pools The discussed priority model is used by a real-time scheduler component newly introduced to Web services middleware. The scheduler ensures the ordered execution of requests based on a pre-defined scheduling algorithm. The algorithm used for scheduling is configurable. Following guideline G2, for this case study we implemented EDF scheduling policy at the dispatcher. As the execution in Web services middleware is carried out by one or more thread pools, the newly introduced real-time scheduler is designed to use a custom made real-time thread pool to manage execution. The scheduler manages execution by enforcing the aforementioned priority model on worker threads.

The scheduler uses Algorithm 2 to reschedule the execution of threads upon the assigning of a new request (N) for execution. All threads with requests currently in execution, are kept track of using a list of references (LT) by the real-time scheduler. The number of requests executing concurrently can be configured and is usually decided on the number of processors available on the server. The worker thread assigned with the request having the earliest deadline at a given time, would be in execution while the others will be queued on TQ, waiting for their turn to re-claim the CPU in the order of their deadlines. The deadline of N is compared sequentially with requests referenced by the members of LT (lines 4–21). If any of the references do not have a request already assigned for execution, the new request is assigned to it immediately and the priority of the worker thread is set to *High*, for it to claim the processor (lines 5–7). If all references have assigned requests, the deadline of N is compared with each of them (line 12). If N has an earlier deadline than any one of them, the worker thread with the latest deadline is preempted by setting the priority to *Mid* (line 13) and subsequently it is queued in TQ for resumption

Algorithm 2 Scheduling of threads

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

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

later (line 14). Thereafter, the reference is set to the worker thread of N and it is allowed to claim the processor by increasing its priority to *High* (lines 15 and 17). However, if the deadline of N is later than that of all requests currently in execution, N is prevented from further execution and is queued for resumption later (lines 23–26). While the rescheduling takes place and tasks with the earliest deadlines are selected, the number of threads must remain unchanged. Access to LT is controlled using a semaphore, to prevent any changes while a scheduling run is in progress. The algorithm acquires a lock on the object (line 3) and releases it as soon as the operations are completed (line 22). Furthermore, the entire Algorithm 2 is marked as a critical section (lines 1 and 27), to prevent any race conditions.

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

Algorithm 3 LT.resetLatestThread implementation

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

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

efficient, the scheduler keeps a special pointer (*ptrLastThread*) directly referencing the thread with the latest deadline, out of all that is in execution. This ensures a quick preemption between N and the target request with the latest deadline. Once the preemption is complete the *ptrLastThread* needs to be reset to reference the worker thread with the latest deadline. In Algorithm 2, this step is carried out after N starts execution (lines 8 and 16). Algorithm 3 carries this out by a sequential comparison of deadlines resulting in a time complexity of $O(n)$, where n is the number of worker threads allowed for execution at any given time. This results in Algorithm 2, also having a similar overall time complexity of $O(n)$.

Figure 6 illustrates the design of the real-time thread pool and the real-time scheduler component. The thread pool, which is an instance of the *RTThreadPoolExecutor* class contains worker threads which are objects of the *RTWorkerThread* class that inherits from the RTSJ *RealtimeThread* class. *RTThreadPoolExecutor* also contains an instance of the scheduling algorithm *RTScheduling*, used for scheduling

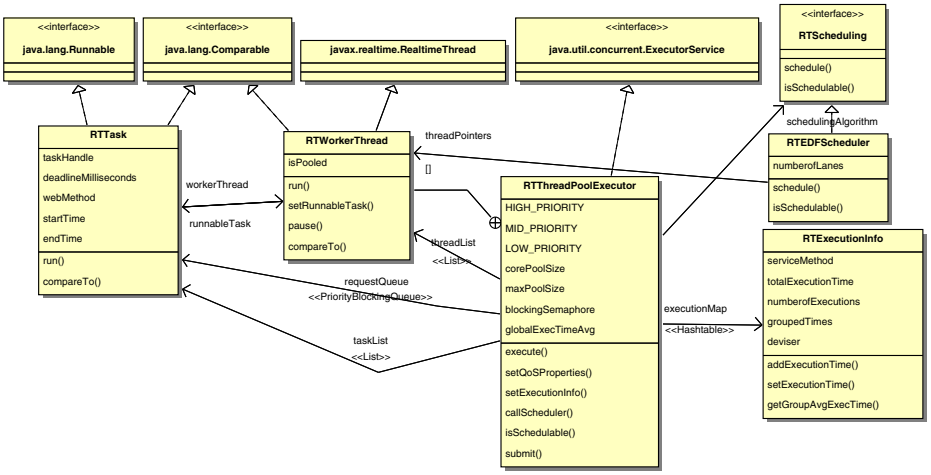

Figure 6 Real-time thread pool class diagram.

the execution of worker threads. For the case study presented, we used an instance of *RTEDFScheduler*. The scheduler controls the execution of worker threads through the *RTThreadPoolExecutor*, by enforcing the aforementioned priority model. Requests are internally represented as instances of *RTTask*, which are assigned to a *RTWorkerThread* on creation. *RTExecutionInfo* instances store summarised execution time history that is used for the schedulability checks done by the scheduler. The level of summarising can be configured and *RTExecutionInfo* instances are stored in a hashtable allowing constant time access and storage. The *RTThreadPoolExecutor* uses *threadList* to keep track of all worker threads and *taskList* to keep track of all request (represented by *RTTask*) instances in the system. Requests are handed over to the threads using a priority based blocking queue (*requestQueue*). Natural ordering of requests in the *requestQueue* upon insertion, is facilitated by implementing the *java.lang.Comparable* in *RTTask*. All three of these data structures are made thread safe, and accessing them is managed using concurrency constructs.

4.4 Implementation details specific to Axis2

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

Figure 7 summarises the specialised enhancements made to Axis2. Both thread pools were replaced with real-time thread pools. To take advantage of multi-core/multi-processor hardware, the executor thread pool was configured to have $n-1$ execution lanes (where n is the number of cores/processors on the server). Therefore, at a given time the requests with the $n-1$ earliest deadlines will get executed. For illustration purposes, Figure 7 contains three execution lanes, each with the currently active thread highlighted. Both thread pools were set to pre-create the worker threads at system start-up to avoid the overhead in thread creation. The functionality of the thread pools are managed by the newly introduced real-time scheduler component. As observed, it manages the execution of all worker threads across all functional modules within the enhanced version of Axis2 (RT-Axis2), using an EDF based scheduling algorithm. The sequence of events inside RT-Axis2 when a request is received, until the completion of its execution is presented in two sequence diagrams. Figure 8 summarises the events that take place in the scheduling phase and Figure 9 the post scheduling phase.

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

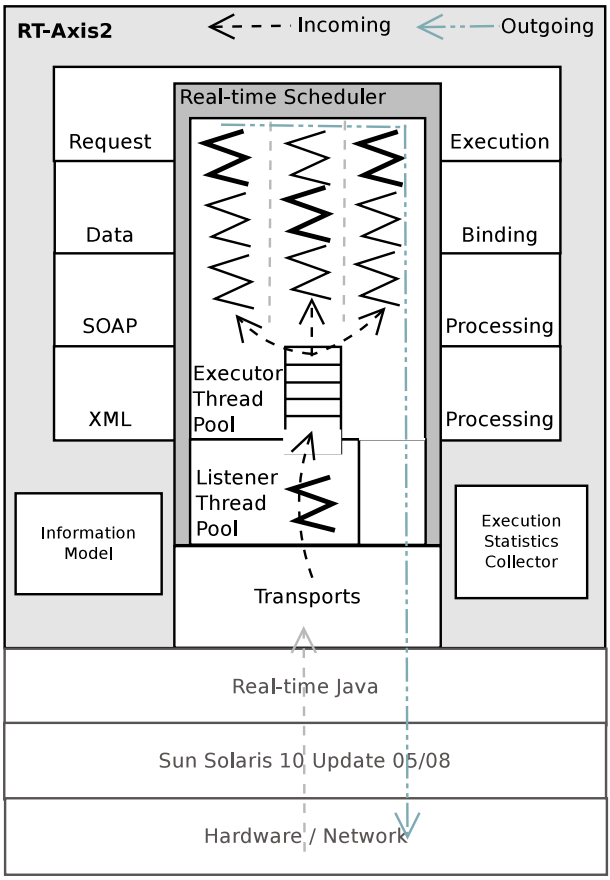

Figure 7 RT-Axis2.

of an extended Axis2 Information Model. When the execution continues onto the SOAP processing module, identification of the request is done and any metadata requests such as for WSDL documents would have the real-time scheduler demote the worker thread to a *Low* priority. If the request is identified to be a service invocation, the schedulability check is carried out immediately using the deadline details available in the Information Model. Furthermore, execution time history information available through a newly introduced Execution Statistics Collector module is internally used by the real-time scheduler to conduct the check. The schedulability check takes place within the SOAP processing module. If the check fails, further processing of the request is suspended immediately and the client is notified using the already built-in fault mechanism of Axis2. Conversely, if the new request can be scheduled, the scheduler immediately conducts a rescheduling of threads, upon which at most only the execution of a single request out of all active, will be interrupted (following aforementioned Algorithm 3). The execution continues onto a normal service invocation process where the results would be conveyed back to the client. Once entire processing of the request is completed, housekeeping activities such as updating execution time history records with times obtained from the current invocation, takes place in the Statistics Collector Module,

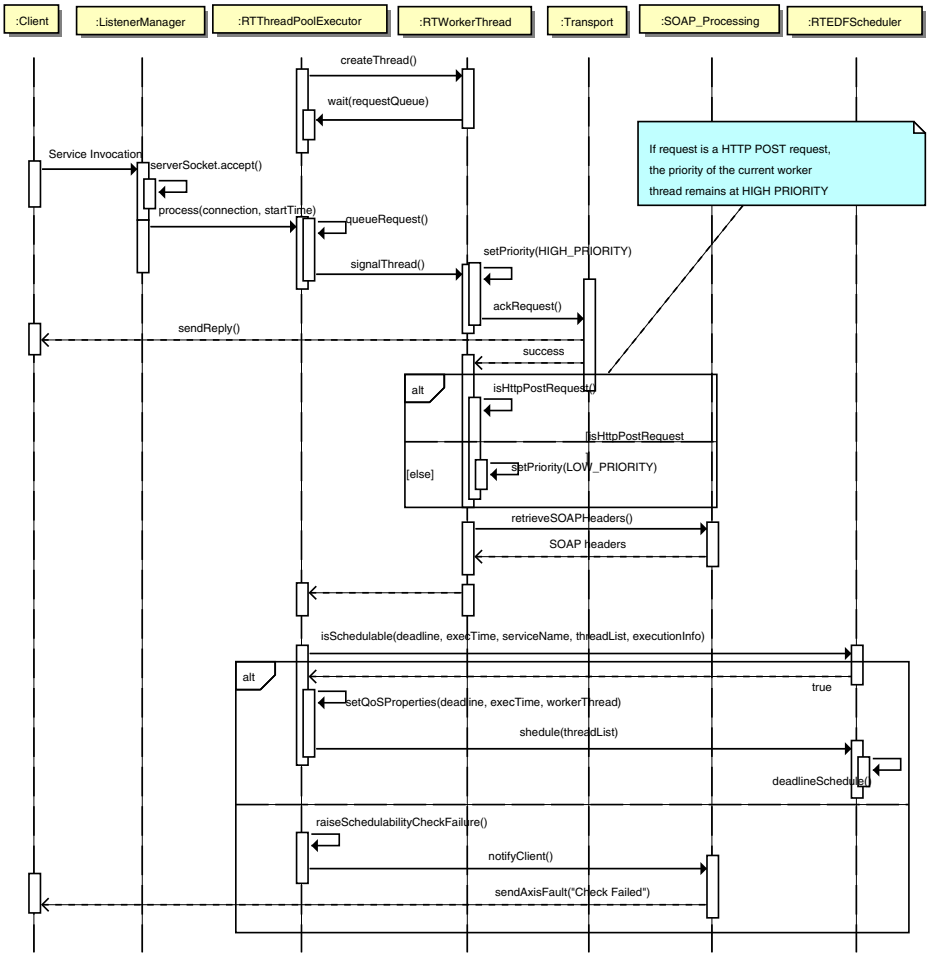

Figure 8 RT-Axis2 execution—sequence of events—scheduling phase.

prior to the worker thread returning back to the pool. Note that time spent on housekeeping activities is considered as subsumed within the overall execution time of a request. This time period is kept track of as a global average and added to the execution time at the time of updating the history records. Once a worker thread completes the assigned request, the scheduler would re-assign it with the next request at the head of the queue for execution.

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

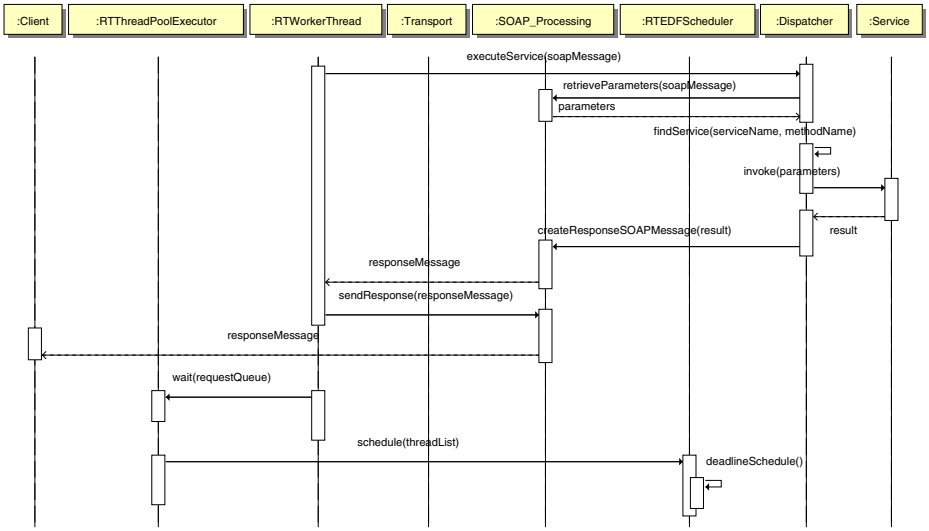

Figure 9 RT-Axis2 execution—sequence of events—post scheduling phase.

4.5 Generic enhancements made to synapse

While the generic enhancements presented earlier are applicable to clusters hosting Web services, in this section we present two dispatching algorithms that are based on those enhancements, that enable a cluster to achieve predictability of execution. The next section, details how these algorithms were implemented in an actual cluster setup.

The dispatching of requests in a cluster is the most important activity in achieving perceived performance levels for request processing. A myriad of dispatching techniques and algorithms optimised on various aspects of performance are used in clusters. However, none of them are designed to guarantee predictability in execution times of requests. In Algorithm 4, we present a simple round-robin scheduling algorithm modified to have the additional step of ensuring predictable execution times. Algorithm 5 is a class based algorithm where requests are divided into classes based on some pre-defined condition and scheduling is done based on the priority class. More details of our work in achieving predictable execution times in clusters serving Web services could be found in [8].

RT-RoundRobin Round-robin scheduling is a commonly used simple dispatching technique used in clusters where requests are evenly distributed among cluster members. RT-RoundRobin retains this feature while having the additional step of subjecting a request to a schedulability check with the selected executor, prior to dispatching. The algorithm works by keeping track of the last executor, a request was assigned to (L) and assigning the next request to a different executor in the cluster. Upon exhausting the list of executors after a complete round of assignments the index used to keep track of the executors (*lastExecIndx*) is reset to the beginning (lines 3–5). Otherwise, the position of the next executor is selected by simply increasing the index (line 6). Using *lastExecIndx*, a reference to the next executor

Algorithm 4 RT-RoundRobin

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

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

Algorithm 5 RT-ClassBased

Require: New request R, List of Executors E, Request Class Mapping RC, Exec-Time History Info H
Ensure: R assigned to an executor or rejected

```
1: Enter Critical Section
2: if H.hasHistoryInfo(R.getParameters) then
3:     T ← H.getExecTime(R.getParameters)
4: else
5:     T ← H.getGlobalAverageExecutionTime
6: end if
7: C ← RC.getRequestClass(T)
8: nextExec ← E.GetExecforReqClass(C)
9: S ← IsSchedulable(nextExec)
10: if S is true then
11:     Assign R to nextExecutor
12: else
13:     Reject R
14: end if
15: Exit Critical Section
```

is obtained (line 8) and the schedulability of the new request is checked against that executor (line 9). If the request is schedulable, it is dispatched for execution (lines 10–12) and the executor is kept track of in L, as the last successful assignment (line 11). A failure in the schedulability check results in the request being rejected (line 14). Objects representing executors are kept in a data structure with constant time access when the index is used. However, the algorithm results in a worst case

time complexity of $O(n)$, due to the resultant complexity of the schedulability check. Although the request to executor matching can be considered as content blind due to the round-robin nature of the algorithm, it results in reduced request arrival rates at each executor. Coupled with the schedulability check that creates a large range of laxities at each executor and deadline based scheduling, the cluster is able to achieve predictable execution times for requests accepted. The complete algorithm is marked as a critical section (lines 1 and 16) to control concurrent executions within.

RT-ClassBased Various methods of class based scheduling are widely used in clusters for content-aware dispatching. For instance, requests belonging to classes A, B and C might be allocated processing time on a 3:2:1 ratio, where for every three requests of class A, two requests of class B and a class C request is scheduled for execution. Likewise, RT-ClassBased dispatches requests between the executors in a cluster, based on their classes with the additional step of guaranteeing a deadline for request execution. In this research, we divide requests into classes based on their size or amount of processing time they require. Each server is tasked with executing a range of task sizes following a mapping done offline made available to the algorithm. As previously discussed, execution time information is available both as history data or as a global average per service, when history data is unavailable. On arrival, the size of the request is obtained using the execution time history if available (lines 2 and 3) or by using the global average instead (line 5). Using the size, the class of a request is obtained through the offline mapping information (line 7) and an executor is matched using the class of the request (line 8). Thereafter, the schedulability of the request on the mapped executor is checked (line 9) and the request is dispatched to it on success (lines 10 and 11). A failure would result in the request being rejected (line 13). The additional information used in the decision making process such as execution time history, request class and executor mappings are stored in data structures with constant access time. This results in the algorithm having a worst case time complexity of $O(n)$ due to the use of the schedulability check in its operation. Furthermore, preventing any race conditions the entire algorithm is marked as a critical section. All data structures used herewith are thread-safe by design.

Note that both these dispatching algorithms check the schedulability of a request with a single executor. Requests with deadlines that cannot be met on the target executor are rejected by the system. We have also developed a dispatching algorithm which considers schedulability of a request with multiple executors. More information about the algorithm and other dispatching techniques can be found in [8].

4.6 Implementation details specific to synapse

Apache Synapse is a lightweight ESB product widely used for enterprise integration in service oriented computing. Designed for message mediation, it is optimised for throughput and processes requests in a *best-effort* manner. ESBs differ from the typical Web services middleware where services are hosted, as they mainly function as message exchanges and gateways where transformations between multiple protocols are supported. The architecture of Synapse is based on Axis2 and contains the Axis2 engine in its core. In processing messages, services such as XML processing and SOAP processing are facilitated by the Axis2 core. Similar to the design of Axis2,

Synapse has an extensible architecture. In and out flow of messages through Synapse could be influenced by programmers using this framework which are modelled as *Sequence* and *Endpoint* objects. Due to such message mediation features, Synapse was an ideal candidate as a dispatcher in a cluster hosting Web services.

Synapse has been designed with throughput in mind and employ several thread pools for its operations. As illustrated in Figure 10, all thread pools were replaced with our real-time thread pools presented earlier as part of the enhancements. All real-time thread pools pre-create worker threads to avoid any delays in object creation. The listener thread pool and the executor pool were configured with $n - 1$ execution lanes where n is the number of cores/processors within the server. The sender pool is configured to have a single worker thread, which is the default in Synapse. Replacing the Axis2 core used by Synapse, with a RT-Axis2 core automatically enables Synapse to have the capabilities such as access to deadline information conveyed through SOAP headers, extraction of deadline information in the XML processing modules and differentiation of request types. A newly introduced real-time scheduler component manages the execution of all worker threads within Synapse throughout the lifetime of a request. Synapse uses the same information model used in Axis2 and as a result, it was easily replaced with the modified version used in RT-Axis2.

The mediation of messages in Synapse is carried out using an extensible event driven framework, which allows programmers to customise the order and type of mediators used (following the chain of responsibility design pattern [6]). This mediator sequencing starts with an event, goes through an arbitrary number of mediators and ends with an endpoint which results in another event. As seen on Figure 11, an incoming message is sent through a sequence of mediators (*in-sequence*) chained together before being dispatched to the URL given by an endpoint in the sequence. Following this design pattern, we implemented a custom mediator (*RT-LoadBalance Endpoint*) and a sequence (Figure 12), that can be configured to use one of the aforementioned dispatching algorithms. It makes use of a standard Synapse *Addressing Endpoint* to dispatch the request to the URL of the executor at the completion of the sequence.

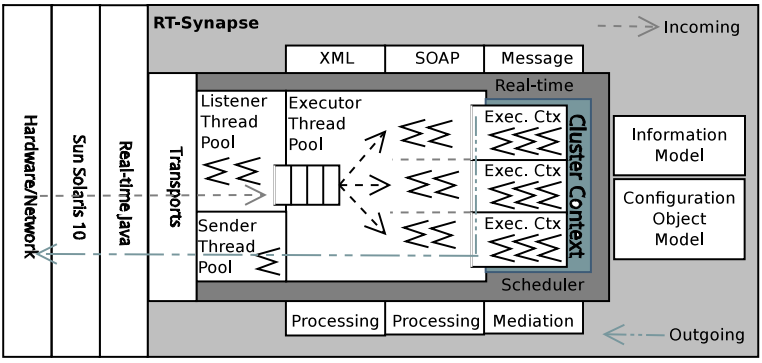

Figure 10 RT-Synapse.

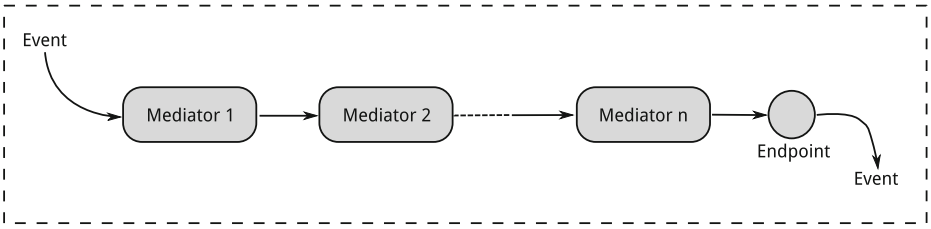

Figure 11 Synapse in-sequence.

In the stand-alone implementation, the server did a multitude of tasks prior to the actual service invocation. An important design decision made for our cluster implementation was to keep the interruptions to the processing that happens at an executor, to a minimum. By design, all the necessary pre-invocation processing (such as schedulability checks and rescheduling of request execution on arrival of a new request) is done at the dispatcher. Thus, the state at each executor and the overall cluster is kept track of at the dispatcher. As illustrated in Figure 13, state of an executor is stored in an *ExecutorContext* instance. State of the overall cluster is kept in a *ClusterContext* instance.

Three ordered queues (based on increasing deadlines) are used to queue requests assigned to an executor. Requests are represented in the system as RTTask instances. Requests waiting to be executed are queued in *WaitingQueue* within the *ExecutorContext*. Once a request is dispatched for execution, its representation is queued in a *SubmittedQueue*. Finally, requests that are preempted from execution are queued in a *PreemptedQueue* at the executor.

Any accepted requests with deadlines later than that of the request currently at the executor, are queued in *WaitingQueue* until their turn for execution. At the completion of a service invocation at the executor, the result is returned to the client via the dispatcher, where the head of the *SubmittedQueue* will be removed at the same time. Moreover, the execution of any requests waiting in the *PreemptedQueue* is resumed and completed in the order of their deadline. Similarly, if the *SubmittedQueue* becomes empty at the completion of a request, the request at the head of the *WaitingQueue* is removed and dispatched for execution, while its representation (the *RTTask* instance) is queued in *SubmittedQueue*. Using this design ensures, the processing of a request at an executor is only interrupted by the acceptance of a request with an earlier deadline. Figure 14 summaries the sequence of events when a request is dispatched and Figure 15 illustrates the design of the classes

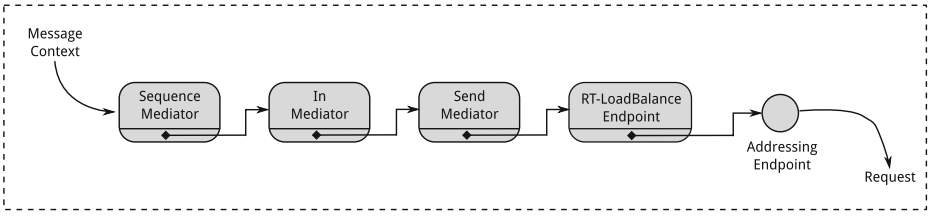

Figure 12 RT-Synapse in sequence.

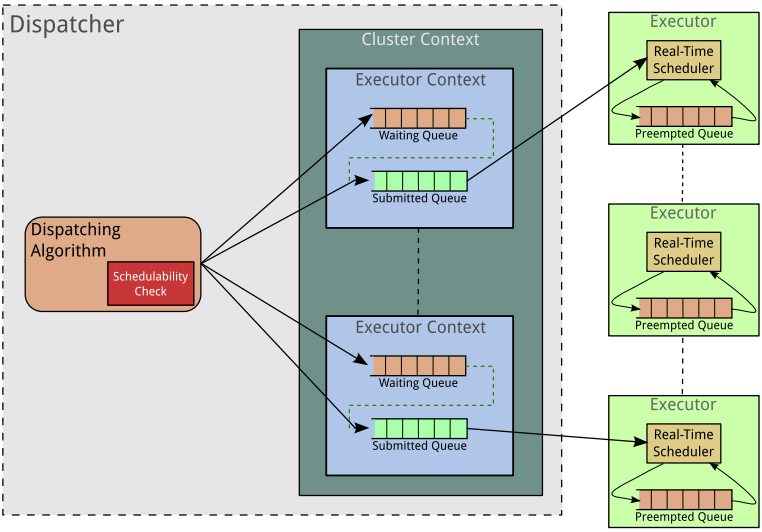

Figure 13 RT-Synapse internals.

introduced into RT-Synapse. As previously mentioned, *RTLoadBalanceEndpoint* is a custom mediator endpoint instance implementation. It makes use of a dispatching algorithm through *RTLoadBalanceAlgorithm* interface by using the *getNextEndpoint()* method. Algorithm specific information that is required at runtime is stored within the *RTLoadBalanceEndpoint* using a *DistributedAlgorithmContext* instance. Moreover, the cluster information is also kept within the endpoint using instances of *ClusterContext* and *ExecutorContext*. *RTTask* instances represent requests within RT-Synapse and it implements the *java.lang.Comparable* interface which would be put to use within the three ordered queues used within the *ExecutorContext*.

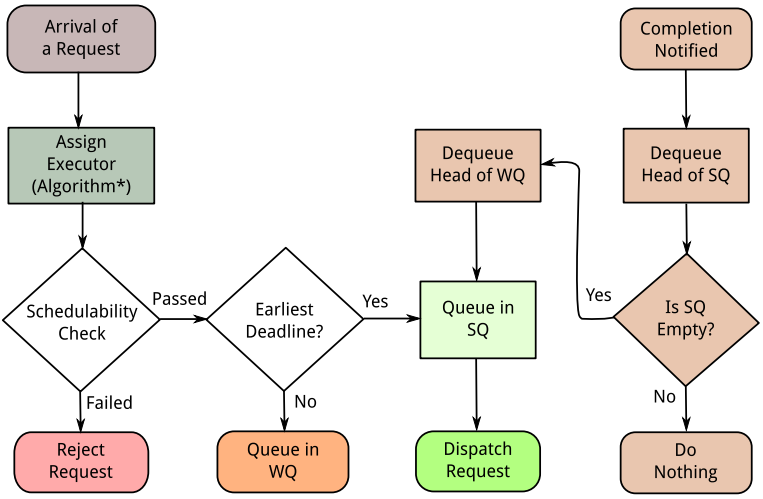

Figure 14 RT-Synapse functionality.

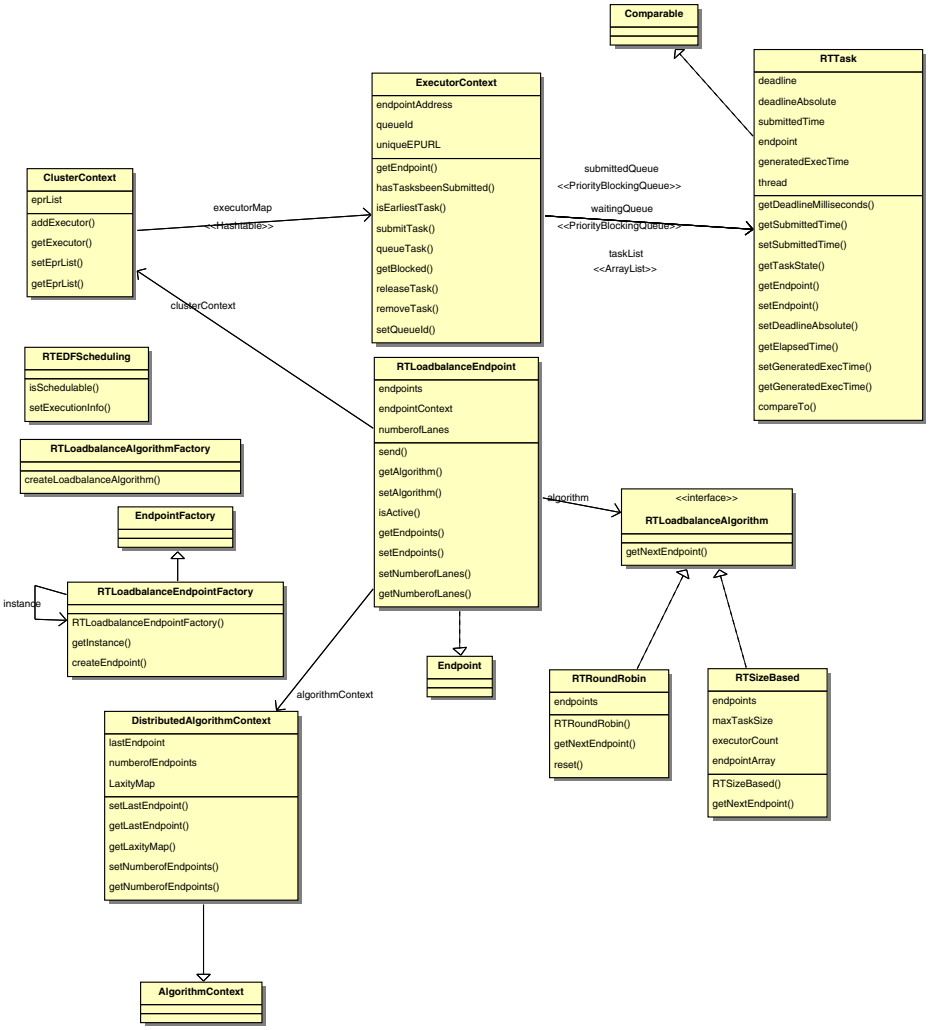

Figure 15 RT-Synapse internals class diagram.

4.7 Minimising priority inversions

Although the techniques we used to minimise priority inversions are product independent, we chose to discuss them separately as they are common to both products used. Priority inversions could impact the execution of a request in two ways. Firstly. a scenario leading to a priority inversion would naturally incur an unwarranted delay in the execution of a request. Following G5, activities that may lead to priority inversion scenarios such as on-screen reporting of operation statuses, recording of output into log files were delayed until the actual request execution is completed. Recording or logging such messages were made using an in-memory model with delayed write, where buffers corresponding to such activities records the output as

and when it happens and direct it only to the intended physical medium such as a file on disk at the end of a complete request execution cycle. This successfully prevented any delays being incurred by such activities on the request execution.

Secondly, priority inversions may result in an unexpected sequence of process execution which portrays a different view of the system activities than actually intended. For instance, a common debugging technique is the use of trace messages either on screen or written to a log file. When debugging the application, such trace messages will result in priority inversions where the sequence of events logged, will not be the actual sequence if not for the trace message itself. Therefore, such trivial debugging techniques cannot be used in the development phase of these systems. Instead, specialised tools such as the Thread Scheduling Visualiser [24] and memory based logging techniques that do not result in priority inversions have to be employed.

5 Sample scenario

The core of our solution is formed by the laxity based selection of requests by the schedulability check and the deadline based execution of the selected requests. Next we illustrate the schedulability check and EDF based scheduling working in unison using an example. For brevity, only a summary is given here. The complete example is presented in the Appendix.

At the start the system has no tasks. Requests arrive at the system in the order mentioned in Table 1 at the respective start times mentioned. On arrival at the system, each request is subject to the schedulability check and either accepted or rejected depending on the outcome. The final column contains the laxity of the request calculated based on its deadline and execution time requirement.

Figure 16 illustrates the schedule achieved by the system for accepted requests and their order of execution. The order of completion achieved by the system has been T3, T4, T7, T2 and T1. It can be observed that at the arrival of request T2 at the system, T1 was already in execution. T2 was accepted by the schedulability check as accepting it does not compromise the deadline T1. However, due to its earlier deadline, T2 was given the processor immediately for execution by preempting (thereby delaying the execution of) T1. Similarly, the schedulability check results positive for request T3 as its deadline could be met while still meeting deadlines of T2 and T1 respectively. Although T4 arrives at the system 1ms after T3, its execution is delayed until the completion of T3 due to its later deadline. Requests T5 and T6 are rejected by the schedulability check, as the acceptance of T5 would result in T4

Table 1 Properties of requests.

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

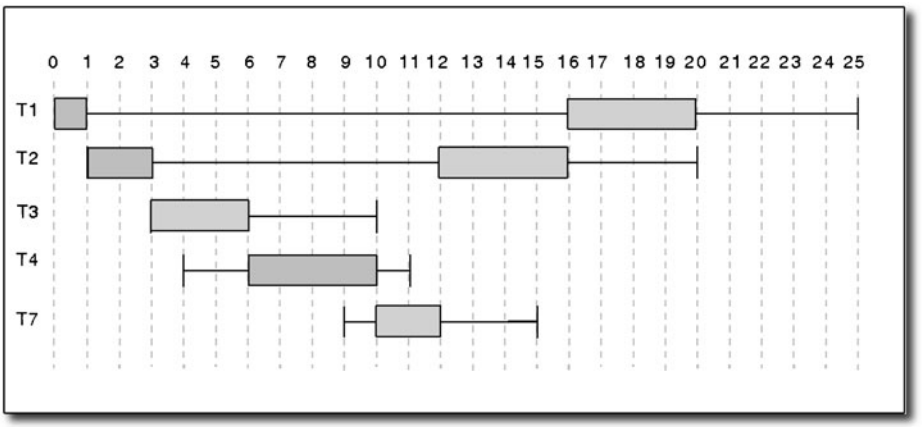

Figure 16 Completed schedule of all accepted requests.

missing its deadline and the acceptance of T6 would result in T2 missing its deadline. Both these incidents occur due to the lower laxity value in T5 and T6. When request T7 arrives, the execution of T4 is almost complete and due to its comparatively large (to T5 and T6) laxity, commencing its execution can be delayed without any deadline misses by the currently active requests. Note that the executions of T3, T4 and T7 were only possible due to the large laxities of T1 and T2 and their own laxities being compatible with them. It can be observed that the resultant laxities of requests accepted for execution are of a wide range of values.

6 Empirical evaluation

To evaluate the validity of the guidelines formed and the resultant enhancements to the two Web services middleware products, an empirical evaluation was conducted comparing the enhanced systems with the unmodified versions to measure the gain of predictability. Although in reality, such systems may be exposed to mixed streams of requests with only a portion of them having hard deadlines, for the evaluation we measure the performance in the most extreme case of the entire request stream carrying hard deadline requirements. Both the stand alone setup and the cluster were exposed to request streams with highly variable request sizes and different arrival rates. Task sizes and arrival rates used, follow a uniform distribution and we use a Web service that allows us to create different sized workloads on the server with the input parameters used. The metrics used for the comparison are the percentage of requests accepted for execution and the percentage of deadlines met out of the accepted requests. Whilst the unmodified version of the middleware does not employ any admission control mechanisms such as the schedulability check, it may reject requests while being under overloaded conditions.

The following hardware and software were used for the two test environments. For the stand-alone setup, both the enhanced version and the unmodified version of Axis2 were deployed on servers with dual Intel Core 2 Duo 3.4 GHz processors (4 cores in total) with 4 GB of RAM, Gigabit Ethernet port running Sun Solaris

Table 2 Performance comparison of unmodified Axis2 vs. RT-Axis2.

Inter-arrival times (sec)	Unmod. Axis2		RT-Axis2	
	% Acc.	% D. Met off % Acc.	% Acc.	% D. Met off % Acc.
0.25–2	100.0	36.2	96.7	100.0
0.25–1	62.4	18.3	58.6	100.0
0.1–0.5	55.1	9.1	30.7	99.7
0.1–0.25	28.7	8.8	18.1	96.7

10 update 05/08 with RTSJ version 2.2. RT-Axis2 is configured with three lanes of execution with 100 worker threads for the stand alone deployment. The cluster setup uses five machines with the same hardware configuration. Four of them run the Executor version of RT-Axis2 (no schedulability check in the execution process) and RT-Synapse acting as the dispatcher is deployed on the other. RT-Axis2 executors are configured with three lanes of execution with 30 worker threads in each lane and RT-Synapse is also configured to have three execution lanes with 30 threads per lane.

Realistic values were used as deadlines in the experimental evaluations. For this purpose, we profiled the Web service for a range of input parameters and derived a functional relationship between the input values and the resultant execution time. The deadline for each task was calculated as a multiplication of the execution time by a random value ranging from 1.5 to 10. The deadline is passed on to both Axis2 and Synapse using SOAP headers.

6.1 Stand-alone middleware—predictability

Table 2 and Figure 17 summarises the comparison between RT-Axis2 and the unmodified version of Axis2. Due to the unconditional acceptance of requests, unmodified Axis2 surpasses RT-Axis2 in the percentage of requests accepted for execution. The schedulability check in RT-Axis2 finds less requests accepted due to their laxity consideration and the deadline requirement. However, both systems accept less requests with decreasing inter-arrival times as the requests arrive at the

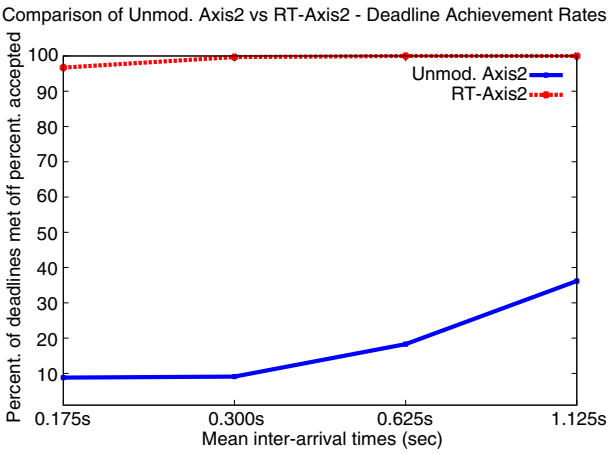

Figure 17 Axis2 and RT-Axis2—deadline achievement rates.

system far more rapidly. This leads to overloaded conditions in unmodified Axis2 that results in requests being dropped. Moreover, due to the *best-effort* nature of request execution, unmodified Axis2 results in unprecedented execution times. This can clearly be seen in the top two graphs of Figure 18 which shows the median execution times resulting from both systems. This phenomenon leads to majority of the deadlines being missed in unmodified Axis2. The schedulability check prevents RT-Axis2 from having such overload conditions and thereby prevents any impact on the execution of accepted requests. Together with deadline based scheduling, RT-Axis2 achieves more than 96% of the deadlines at all times, outperforming Axis2 in the evaluations. Comparing resultant execution times in Figure 18, it can clearly be seen that the range of values achieved by Axis2 is far greater compared to the range achieved by RT-Axis2. This is a clear example of the unpredictable nature of *best-effort* execution in such Web services middleware. Furthermore, the two graphs in the second row shows the resultant execution times sorted by the task size, it can clearly be seen that the fluctuation of execution times are far greater for large task sizes. Whilst some fluctuation exists even in RT-Axis2 execution times, they are smaller controlled delays based on the laxity of a request.

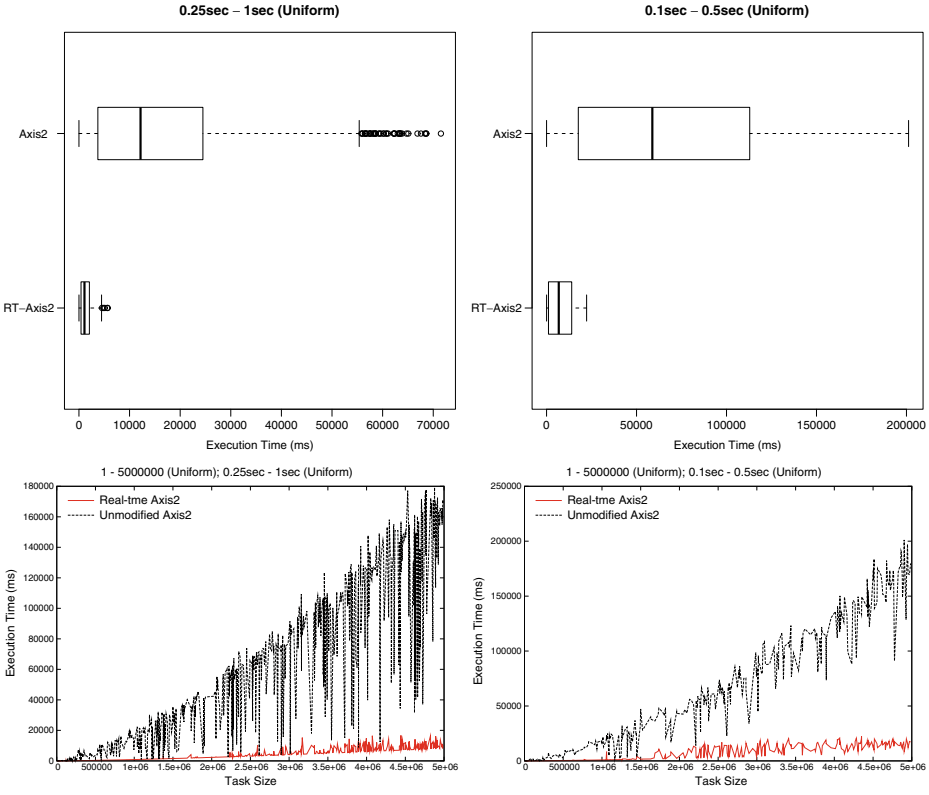

Figure 18 Execution time comparisons—Axis2 and RT-Axis2.

6.2 Cluster based middleware—predictability

Evaluating the cluster based setup, the performance of the enhanced cluster is compared with an unmodified cluster for the performance of both round-robin and class-based dispatching algorithms. The unmodified cluster dispatches and executes requests in a *best-effort* manner. The evaluation is done for various arrival rates, gradually increasing the number of executors in a cluster starting from 2 upto a maximum of 4.

6.2.1 Round-robin dispatching

The round-robin dispatching scenario is a fair evaluator for the level of predictability, that could be achieved by enhancements made to a cluster based middleware setup, following the guidelines. We compare the performance of RT-RobinRobin with simple round-robin dispatching using the same cluster setup. With the latter, there is no conditional acceptance of requests and execution happens in a *best-effort* manner. RT-RoundRobin uses the schedulability check to select requests for execution based on their laxity and selected requests get executed in the order of their deadlines. Table 3 summarises the results for the round-robin runs and Figure 19 summarises the results graphically (Note that the *x-axis* of the graph contains the mean values for the respective inter-arrival time periods mentioned in Table 3). Due to the unconditional acceptance of requests, simple round-robin results in higher request acceptance rates. Herein, the rejection of requests as the inter-arrival times decrease were caused by overloaded conditions resulted in the cluster. While RT-RoundRobin results in lower acceptance percentages comparatively, it clearly outperforms simple round-robin in the resultant percentage of deadlines met. The best performance simple round-robin could achieve is 51.5% of the deadlines with almost all requests being accepted for execution, when 4 executors were used in the cluster. However, RT-RoundRobin, consistently achieves more than 90% of the deadlines in all the runs conducted, while maintaining decent acceptance rates. Although a higher number of requests are accepted for execution with simple round-robin scheduling, the executors get overloaded as a result of *best-effort* execution of requests. The overloading leads to the system being stalled and the overall execution of requests being delayed, while other requests ready for execution are made to wait at the dispatcher. As illustrated in Figure 20, this results in longer execution times and comparatively a very large range of values in simple round-robin scheduling, which result in higher rates of deadline misses.

6.2.2 Class-based dispatching

In the class based dispatching scenario, we use request size to be the criteria for classification. Herein, segregation of requests based on size is a widely used technique that reduces the overall waiting time of the system. RT-ClassBased makes use of this feature whilst introducing the additional step of selecting requests for execution based on their laxity and execution of requests based on their deadlines. It is compared with a trivial class-based scheduling algorithm where each executor is

Table 3 Performance comparison of round robin vs. RT-RoundRobin.

Inter arrival times (sec)	Round robin (non real-time)						RT-RoundRobin					
	2 executors		3 executors		4 executors		2 executors		3 executors		4 executors	
	% Acc.	% D. Met	% Acc.	% D. Met	% Acc.	% D. Met	% Acc.	% D. Met	% Acc.	% D. Met	% Acc.	% D. Met
0.25–1	99.5	28.8	99.8	37.2	99.9	51.5	88.0	99.0	99.0	100.0	99.9	100.0
0.1–0.5	62.3	20.3	89.0	28.4	98.0	39.7	52.0	96.4	74.0	99.0	99.4	99.9
0.1–0.25	49.0	15.0	67.3	20.0	74.1	33.2	28.0	96.0	47.0	97.6	78.0	99.0
0.05–1	38.8	6.3	52.6	9.1	68.0	13.6	20.5	90.0	37.5	95.0	46.3	99.0

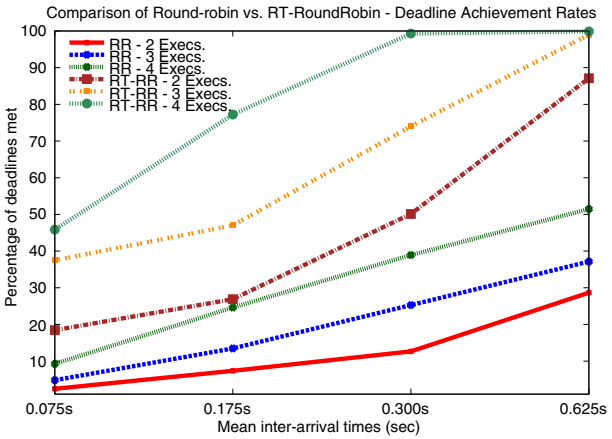

Figure 19 Execution time comparisons—RR vs. RT-RR.

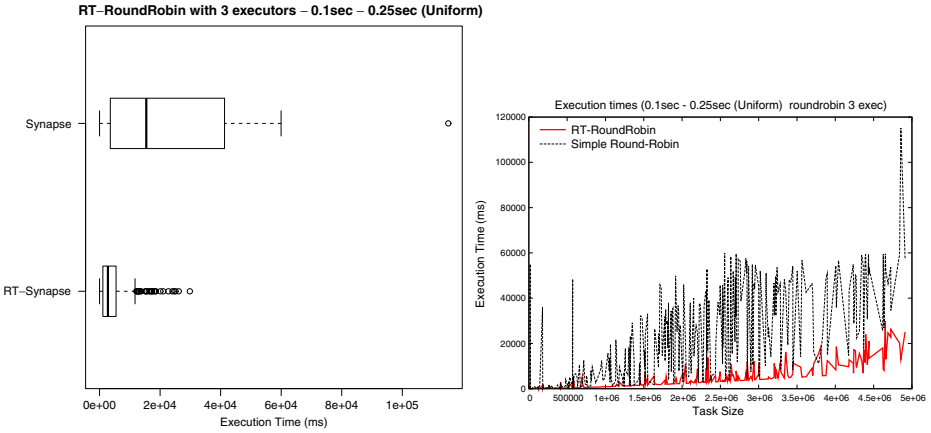

Figure 20 Execution time comparisons.

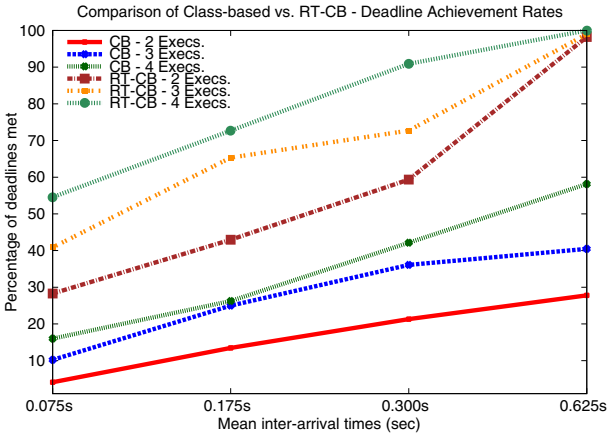

Figure 21 Execution time comparisons—CB vs. RT-CB.

assigned with a request size range. Table 4 contains the results while Figure 21 summarises them graphically (Note that the *x-axis* of the graph contains the mean values for the respective inter-arrival time periods mentioned in Table 4). The size based segregation of requests prevents scenarios where requests with a large size disparity compete for the same processing resource. Such scenarios would have requests with shorter execution times being queued behind requests with longer execution times. As observed from the results obtained, class-based scheduling performs better than round-robin scheduling due to this reason. In Figure 22, the segregation of request sizes among three executors, between round-robin and class-based scheduling is clearly visible. The dispersion of the execution times around each task size is less in class-based scheduling, as a result of the lower task size variance at each executor. Moreover, the dispersion of smaller sized requests are much lower in class-based scheduling (compared to round-robin) also due to the aforementioned reason. Similarly, Figure 23 illustrates the resultant CPU utilisation levels for one of the runs with three executors being used. The first graph for round-robin scheduling has all three executors being utilised at similar levels while the second graph shows the different levels of utilisation at the executors due to the size based segregation. Whilst class-based achieves better results against round-robin scheduling, RT-ClassBased performs better when the percentage of deadlines met, are considered. Irrespective of scheduling decisions being made based on the size of requests, unconditional acceptance of requests and *best-effort* nature of execution may lead to overloaded conditions and requests being rejected. Moreover, the sharing of the CPU in *best-effort* processing prolongs the execution of all requests executing in parallel, thereby resulting in deadline misses. The schedulability check in RT-ClassBased coupled with deadline based scheduling, achieves more than 94% of the deadlines in any given scenario. Although acceptance rates are lower than simple class-based, the resultant deadline achievement rates clearly confirms RT-ClassBased outperforming its unmodified counterpart, in terms of predictability.

6.3 Laxity based request selection

As discussed earlier, enhancements made to RT-Axis2 and RT-Synapse results in conditional acceptance of requests, based on their laxities. The introduced schedulability check works by trying to match the laxity of a target request with the already accepted requests that overlaps with its lifespan in the system. A request is accepted based on the compatibility of its laxity with that of already accepted requests, depending on the processor demand within its lifespan in the system. As illustrated in the sample scenario (Section 5), a request with a larger laxity will allow many other requests to be scheduled within its lifespan and a smaller laxity will require the request be scheduled together with other tasks with higher laxities. The nature of this selection process eventually results in a wider range of laxities at a server in any given time period.

Recall that laxity indicates the ability to delay the execution of a request, while still meeting its deadline requirement. It is usually indicated as a ratio between the deadline and the execution time of a request. Figure 24 contains two box-plots showing the range of laxities resulted at each server for the stand-alone and cluster based scenarios. The first graph contains laxity plots for all stand-alone scenarios. Recall that the *best-effort* nature of Axis2 executes as many requests as possible

Table 4 Performance comparisons of class based vs. RT-ClassBased.

Inter arrival times (sec)	Class based (non real-time)						RT-ClassBased					
	2 executors		3 executors		4 executors		2 executors		3 executors		4 executors	
	% Acc.	% D. Met	% Acc.	% D. Met	% Acc.	% D. Met	% Acc.	% D. Met	% Acc.	% D. Met	% Acc.	% D. Met
0.25–1	100.0	27.8	99.2	40.8	99.9	58.2	99.2	99.0	100.0	99.0	100.0	100.0
0.1–0.5	82.0	26.0	98.6	36.6	99.4	42.4	62.2	95.4	76.7	94.8	90.9	100.0
0.1–0.25	74.8	18.0	83.3	30.0	86.9	30.2	45.4	94.6	66.0	99.0	74.4	97.7
0.05–0.1	52.7	7.8	75.6	13.5	78.0	20.5	28.6	98.9	44.7	91.4	55.1	99.0

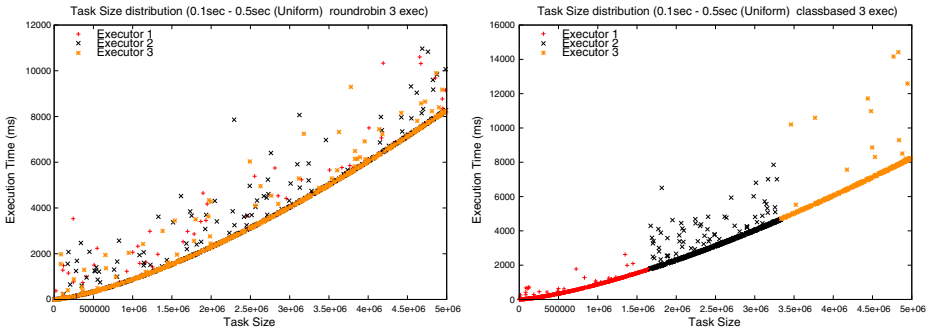

Figure 22 Task size distribution at executors.

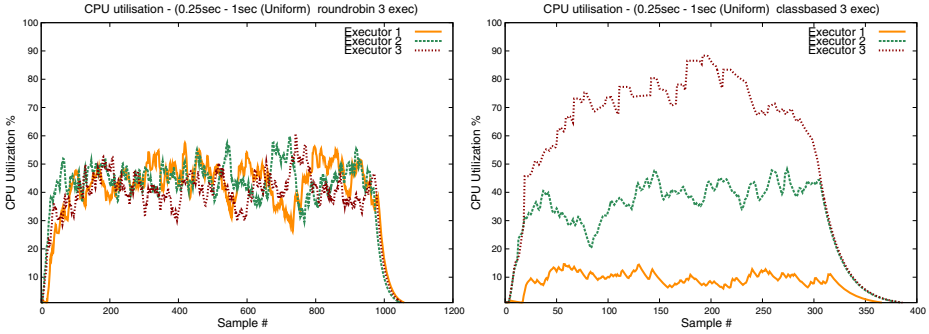

Figure 23 CPU utilisation at executors.

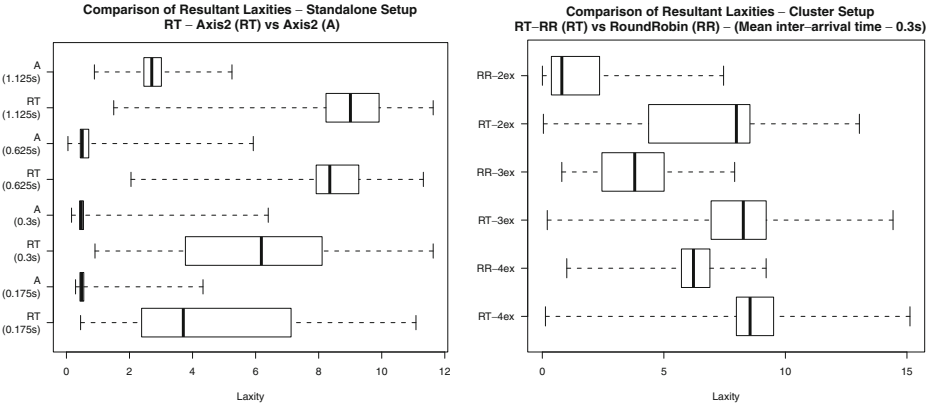

Figure 24 Comparison of resultant laxities from admission control.

in parallel and leads to overload conditions and deadlines misses. These conditions gets worse with short inter-arrival times as seen on Table 2. In every run, many of the initial requests handled by Axis2 get executed very quickly as the competition for CPU is less, till more requests arrive. This results in lower execution times that contribute to higher laxity values. However, as the number of requests increase the *best-effort* processing overloads the server and many of the requests being executed sharing the processor result in execution times past their deadline requirement. With less than 36.2% of the requests meeting their deadlines, majority of the requests in every Axis2 run result in laxity values less than 1. As visible on the graph, the median laxity value gets lower with the increasing arrival rate.

Similarly, the median laxity value decreases with higher arrival rates for RT-Axis2. However, the selection of requests by the schedulability check results in a range of laxities at the server. Finding a complementing laxity schedulable with existing requests, results in this phenomenon which can clearly be observed in the figure above. The runs with slow request arrivals result in a higher median value, and it decreases with increasing request arrivals although still resulting in a large variety of values. This contributes towards requests meeting their deadline requirement even in conditions with high task arrivals.

Similar observations can be made in the cluster based configurations depicted in the second graph of Figure 24. Herein, the box-plots are from the runs conducted with RoundRobin (RR) and RT-RoundRobin (RT) algorithms in the cluster. Across all experimental runs the request inter-arrival time is the same mean value of 0.3 sec. Each plot corresponds to the average laxities achieved at the servers for each algorithm with 2, 3 and 4 executors, respectively. Although similar patterns to the stand-alone results could be observed, the unmodified cluster running Synapse with Axis2 comparatively achieve a higher request acceptance and deadline rates due to the use of multiple executors, albeit higher request arrival rates being used for the experiments. The two executor setup achieves the lowest median laxities due to the higher number of deadline misses of all runs. As the number of executors increase the miss rate decreases and the median laxity value increases. Moreover, the upper bounds achieved by the cluster setup is higher than the single server setup for obvious reasons.

The enhanced cluster setup with RT-Synapse and RT-Axis2 combination demonstrates a similar pattern in the resultant laxity values. Naturally, the selection process achieves a higher variety or range of laxities and the median value decreases with high request arrivals. Although the increase of executors in the cluster does not result in a major change to the median laxity value, a shift in values from the lower quartile to the upper quartile is visible. This is due to the constant rate of deadlines achieved by the setup (96%) and the increasing number of accepted requests being distributed to multiple executors in the cluster. From both configurations, it is clearly visible that such a purposeful selection of requests is a necessity for achieving predictability in execution.

6.4 Throughput comparison

Next, we compare the unmodified and the enhanced versions of both configurations on their throughput. For this comparison we define throughput to be the number of requests processed by a server in a given unit of time. Herein, for the unmodified

Table 5 Throughput comparison of unmodified Axis2 vs. RT-Axis2.

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

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

Table 5 contains throughput values measured (requests per second) for unmodified Axis2 and RT-Axis2 under different request arrival rates, for each scenario discussed earlier. Figure 25 summarises the results graphically. The second column for RT-Axis2 contains throughput measured without considering requests rejected. Axis2 is configured by default with 25 worker threads pre-created at start-up and the ability to create upto 150 worker threads when the request queue is filled up. Practically, it is possible for all 150 threads to be in execution sharing the processor at any given time as there are no differentiation or control over how threads execute. With the highest mean inter-arrival time (1.125 sec), Axis2 records better throughput values compared to RT-Axis2. With increasing arrival rates, Axis2 records decreasing throughput values. The *best-effort* nature of Axis2 contributes to it being overloaded in quick-time and the system being unresponsive, resulting long delays in request completions. Moreover, incoming requests drop out due to unresponsiveness of the system. This condition increases with request arrival rates.

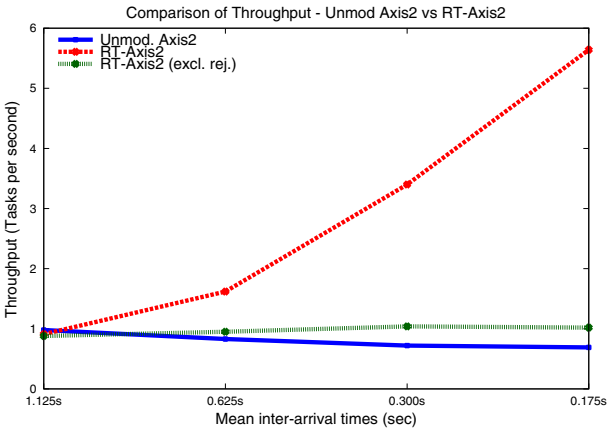

Figure 25 Comparison of throughput—Axis2 vs. RT-Axis2.

The enhancements made to RT-Axis2 enables control over the execution of worker threads. The configuration of three execution lanes and the functionality of the real-time scheduler component, restricts only three threads to be in execution at any given time. There maybe upto 100 worker threads pre-created, ready to be used for request execution or with assigned requests with later deadlines. Their use of the CPU is controlled by the scheduler using lower priorities. In the lowest request arrival configuration, RT-Axis2 achieves a marginally lower throughput value. As request arrival rates are increased, the throughput of the system also increases accordingly. Although request traffic increases, the schedulability check in RT-Axis2 prevents the system from being overloaded. However, a side-effect of this is the increased amount of rejections as observed in Table 2. The processing of a request upto the completion of the schedulability check, is comparatively quicker than the intended service execution. A rejected request will incur only this portion of processing. As a result, the throughput recorded in high traffic conditions comprise of a considerable amount of rejected requests. It is clearly observed in the respective secondary throughput values calculated, excluding the rejections. At the smallest mean inter-arrival time (0.175 sec), the ratio of accepted to rejected requests is around 1:4.

A similar pattern is also visible in the cluster setup. Table 6 contains the results for the cluster running the two round-robin based scenarios for different executor configurations and Figure 26 summarises them graphically. The throughput result achieved reflects the performance of Synapse and RT-Synapse as it is the focal point of the system where all requests and replies converge. When a particular cluster configuration is considered, the aforementioned pattern is clearly visible for increasing request arrival rates. The unmodified cluster with Synapse and Axis2, shows an increase of throughput with increasing request arrivals. However it is unable to continue the increase as the system gets overloaded by requests. Similar to the stand-alone setup, throughput of the enhanced cluster increases with request arrival rates although the higher values of throughput is contributed largely by rejected requests. If request rejections are excluded from the throughput calculations, the system still achieves acceptable throughput rates while preventing it from reaching overloaded conditions. For a particular arrival-rate, observe that throughput does not change with the addition of executors into the cluster. While this adds more processing power to the back-end and leads to more deadlines being achieved, the throughput rate achieved by the dispatcher (be it Synapse or RT-Synapse) remains the same for the given mean inter-arrival time.

6.5 Discussion

Through the empirical evaluation we tried to ascertain the validity of the guidelines provided and the enhancements made accordingly, to Web services middleware for achieving predictability of service execution. The empirical results confirm that predictability of execution could certainly be achieved in both stand-alone and cluster based Web services middleware with the suggestions made. Furthermore, the results obtained in the cluster based evaluations confirm that the suggested enhancements are indeed scalable. The conditions tested for were worst case scenarios of high request arrival rates, all requests having hard deadline requirements and highly variable task size distributions. All candidates the enhanced systems were tested

Table 6 Throughput comparison of round robin vs. RT-RoundRobin.

Mean	Round robin (non real-time)			RT-RoundRobin					
inter-arrival	2 exec.	3 exec.	4 exec.	2 exec.		3 exec.		4 exec.	
time	sec^{-1}	sec^{-1}	sec^{-1}	sec^{-1}	sec^{-1} (excl. rej.)	sec^{-1}	sec^{-1} (excl. rej.)	sec^{-1}	sec^{-1} (excl. rej.)
0.625 sec (low)	1.50	1.51	1.51	1.62	1.42	1.62	1.61	1.62	1.62
0.300 sec	0.81	2.91	2.92	3.31	1.72	3.30	2.44	3.37	3.36
0.175 sec	0.98	1.67	2.30	5.54	1.55	5.46	2.56	5.43	4.24
0.075 sec (high)	1.06	1.40	1.87	10.3	2.11	10.8	4.05	11.1	5.13

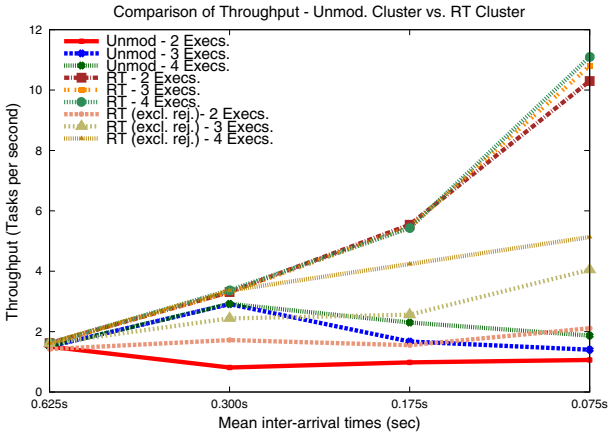

Figure 26 Comparison of throughput—Synapse vs. RT-Synapse.

against, unconditionally accepted requests for execution, that resulted in higher acceptance rates. However, this led them into overload conditions which resulted in the rejection of requests, as non-responsiveness of the system led to request timeouts. Moreover, this also led to high rates of deadline misses, 58.2% being the highest achieved and at times being less than 10%. Although the enhanced versions in both scenarios resulted in lower acceptance levels, they excelled in achieving more than 90% of their deadlines in every scenario tested for.

With RT-Axis2, means of achieving predictability of execution in standalone Web services middleware was evaluated. With the empirical results obtained, it is clear that RT-Axis2 outperforms the unmodified version securing at least 97% of the deadlines while maintaining comparable request acceptance rates. The unconditional acceptance of requests in the unmodified version together with the *best-effort* nature of request execution, works well in scenarios where completion time limit is of less importance. With the thread-pools in use executing as many requests as possible in parallel, their completion times increase with the number of requests being executed, where the maximum number of worker threads serve as an upper bound. As seen on Figure 18, the resulting longer execution times contribute to the number of deadlines missed. Moreover, this leads to very high utilisation levels, that could create overload conditions, which result in requests timing out due to unresponsiveness of the server. Conditional acceptance of requests based on laxity in RT-Axis2 ensures that a request can be scheduled together with already accepted requests whilst ensuring all deadlines are met. This is further facilitated by the deadline based scheduling used for request execution. Supported by the priority model introduced, the features available in the development platform and the OS, predictability in execution is achieved successfully. With increased arrival rates, more requests compete for the same window of time. In such scenarios, the schedulability check may reject more number of requests. The rejections could be reduced with the use of a cluster based setup where more than one server is used for request execution.

With the RT-RoundRobin evaluation, the benefits of the predictability enhancements in a cluster are directly visible as round-robin dispatching is a simple technique in its pure form. Moreover, it is also a perfect example of how a simple dispatching technique could be enhanced to achieve predictable execution. Similar to the standalone scenario, RT-RoundRobin clearly outperformed its unmodified counterpart.

The effectiveness of round-robin scheduling is in the even distribution of requests it results in throughout the cluster. However, the same reason makes it unsuitable when predictability of execution has a higher importance. Even distribution of requests create a high variability of request sizes at each executor. Yet, this being a content blind dispatching technique, may result in higher loads and longer execution times due to the variability of request sizes. As the request execution happens in a *best-effort* manner with thread-pools executing as many requests as possible in parallel, all requests will complete with longer execution times albeit the throughput achieved (Figure 20). With RT-RoundRobin, the high variability of request sizes is circumvented due to the laxity based request selection in the schedulability check. Irrespective of the size of the request, a request is selected for execution only if its laxity enables the deadline to be met while being scheduled with already accepted requests. Moreover, this also acts as an admission control mechanism that prevents server overloads. The high variability of laxities resulted by the schedulability check theoretically ensures deadlines of all requests accepted could be met, while deadline based scheduling ensures it practically.

RT-ClassBased algorithm serves as an example of how a request-aware scheduling policy could be enhanced to achieve predictable execution times. As with the other scenarios the RT-ClassBased algorithm outperformed the simple class-based version when deadline achievement rates are considered. As the size of a request was the criteria in matching a request to an executor, size based scheduling ensures that each executor is only faced with requests of similar sizes irrespective of the original task size distribution (Figure 22). This prevents chances of smaller sized tasks having longer waiting times due to the processing of a large sized request. Clearly, task-based scheduling in its pure form had better results compared to round-robin scheduling. RT-ClassBased combines this phenomenon with the additional guarantee of deadline requirement of requests being met. Yet again the difference is in the selection of requests based on laxity by the schedulability check and the purposeful scheduling of requests based on their deadlines. However, class-based scheduling in its pure form performs badly when percentage of deadlines met is considered. Similarly to earlier discussions, unconditional acceptance of requests and *best-effort* scheduling results in unpredictable execution times and deadlines being lost. Performance comparison of these two dispatching algorithms with other techniques that conduct multiple schedulability checks per request, can be found in [8].

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

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

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

7 Related work

Many existing work related to Web service execution could be found in the area of QoS. A common feature that could be observed in many of them is service discovery or composition based on a QoS criteria. Many of them make the assumption that the Web services middleware and the underlying infrastructure used, will guarantee perceived QoS levels within a probabilistic measure. Work by Ran [18] and Tian et al. [25] facilitate the discovery of services based on QoS parameters by incorporating information about QoS levels provided by services in modified Web service directories. QoS brokers facilitate the discovery by liaising with the directory service on behalf of the clients, based on its QoS requirement. Work of Zeng et al. [28] extends this to service compositions where the selection of services for a composite service is based on QoS attributes each service is able to provide. Yet again the guarantee of QoS levels by the middleware is assumed.

While the work mentioned assume QoS levels are guaranteed by the middleware, there are attempts at achieving different levels of quality in the middleware. Work of Sharma et al. [19] introduces few methods of differentiation into the processing of requests. Requests are classified into various service classes based on non-functional attributes such as nature of application (i.e. a stock trading service versus a price lookup service), the device being used as a client (i.e. a Personal Computer versus a mobile device) and nature of client (i.e. paying customer request versus a free request). Priorities are assigned to each service class based on these attributes. The arrival rate of each category is monitored and the priorities are dynamically adjusted using a penalty function to reduce starvation. The penalty function penalises a priority on a lower than normal arrival rate and enforces it positively on higher than

normal arrivals. The solution achieves some level of differentiation in the throughput of the requests and tries to maintain a pre-defined ratio between the service classes. Similarly, the work of Tien et al. [26] classifies requests into service classes based on a pre-defined SLAs between the service provider and clients. Operations are profiled offline and the information obtained is used to calculate the workload required when a service is invoked. A scheduler component in the middleware evaluates the request arrivals and ensures a pre-defined ratio of the service classes, in processing. Although the operations considered in the approach are non-functional properties of the service such as security processing, the same technique could be applied for functional attributes as well. However, the scheduler simply maintains the ratio of the different classes in the number of requests processed, rather than considering the actual execution times resulted. Helander et al. [14] uses SOAP based Web services in an embedded real-time environment where Web services are used for communication between the components in the system. Patterns of communication, sequence of events, their resource requirements and execution times are known *a priori* due to the embedded nature of the system. Therefore, the execution sequences and the schedules are planned out at design time of the system. A statistical model is used to plan for any variations or possible jitter and resources are over reserved to counter such scenarios. The closed nature of the system allows planning ahead, thereby reducing the variation in execution times. However, the solution is not applicable for open systems as in the use of Web services over the internet where request characteristics, arrival patterns and sequences are unknown and unpredictable at design time. Most of the discussed work, maintain a certain ratio of processing between the request classes and achieve perceived execution times in a probabilistic manner. While all attempts achieve some form of differentiation in request processing, none of them can guarantee predictable execution times under any traffic condition. Although Helander et al. [14] uses real-time scheduling techniques in their solution, the requirement of having all information necessary for scheduling at design time makes it difficult to be used in open systems such as on the Internet due to the unpredictable nature of requests.

Similarly, mechanisms of achieving perceived performance levels as outlined in SLAs can also be seen in clusters hosting Web services. The work of Pacifici et al. [17] uses a multi-level dispatching technique where a layer 4 switch acts as the first level dispatcher which distributes requests among several gateways in the cluster in a content-blind manner. Pre-defined SLAs classify requests in to several grades where customers pay to be in a certain grade with a probabilistic guarantee on execution times. Gateways dispatch requests among cluster servers hosting identical content and a global dispatcher keeps track of the server resources used and currently available at each server. A utility function is used by the resource manager to compute the resource consumption and calculate the number of connections from each grade a server could handle in a given window of time. This information is disseminated periodically among the gateways, which make use of them for dispatching decisions. García et al. [9] takes a similar approach where an SLA is used to specify the maximum response times for each service delivered by the provider. Each customer is guaranteed a probabilistic measure of the response times specified in the SLA. Cluster servers host identical content and a monitoring module in each server keeps track of the resource use and request execution. This information is periodically updated at a controller module which compares the information with the perceived

response times on the SLA. Using the calculated statistics, the controller decides on the acceptance of a request for execution upon being queried by the load balancer. Continuation of this process leads to dynamically adjusting the request acceptance to achieve the probabilistic measures of execution times for each client. The work of Gmach et al. [12] takes a different adaptive approach by using fuzzy logic to optimise parameters such as resource availability, execution times for each class of requests and performance levels perceived in the SLA. A management module uses fuzzy logic to calculate the optimal parameters for the servers where requests of certain classes will have priority over others. Similarly, Cao et al. [4] presents a Jini based self-configurable service process engine which dynamically balances the load among services hosted, based on a predefined model. A heuristic technique is used to classify every request, using a tag specifying the workload it would incur on execution. The workload tags are intended for a controller module, which dynamically configures the engine based on this information using a fuzzy control algorithm. The heuristic algorithm calculates the workload incurred by a request, using a set of probability based values for each function the engine must perform to complete the request. The fuzzy control function maps this value into a generalised fuzzy number based on predefined functions that classify a request based on its resource requirement. Depending on the projected workload the control module dynamically increases or decreases the active service instances to maintain the perceived level of performance. All these approaches discussed considers service execution time as a QoS parameter and try to achieve pre-determined levels of performance. As it is a probabilistic measure, none of them can guarantee it in a consistent manner.

Similar attempts at achieving different levels of performance through service differentiation could be seen in serving simple Web requests. Eggbert et al. [5], introduces an application level service differentiation to a Web server with two service classes. Processes that serve the Web requests are grouped into one service class named foreground processes and others such as cache managers that uses speculative pull and push transactions are categorised into the another as background processes. Foreground processes are comparatively more CPU bound while background processes are more network bound. The work presents three different backgrounding mechanisms that allocates processing resources between these two classes in different ratios, thereby controlling the number of processes from both classes that use the CPU. Kihl et al. [15] presents an admission control mechanism for Web servers which rejects requests based on load conditions at the server. Their solution uses a combination of queueing theory and control theory to ensure the CPU utilisation is preserved at a certain level. Queueing theory is used to model an Apache Web server as a GI/G/1 system and the control theory is used to decide on the number of requests accepted. While these attempts succeed at introducing service differentiation or selective request execution, none of them have all the components needed to ensure consistent predictable execution times. Moreover, being application level solutions they lack support from the system level, that would ensure the required level of consistency.

7.1 Summary

In the work discussed for both stand-alone and cluster based Web services, a commonality is the consideration of execution time as a QoS parameter. Various

Table 7 Compliance of related work to predictability requirements.

Req.	Sharma et al. [19] Tien et al. [26]	Pacifici et al. [17]	Gmach et al. [12]	Cao et al. [4]	García et al. [9]	Helander et al. [14]	Our method
G1	⊗	⊗	⊗	⊗	⊗	⊗	●
G2	⊗	⊗	◐	⊗	◐	●	●
G3	⊗	◐	⊗	⊗	◐	◐	●
G4	●	●	●	●	●	◐	●
G5	⊗	⊗	⊗	⊗	⊗	⊗	●

⊗ not compliant, ◐ partially, ● fully

approaches are taken to achieve some level of differentiation in request processing in all of them and many try to achieve a probability based measure of execution time among different classes of requests. Many of them try to dynamically adjust the ratio of request processing to meet the pre-defined levels of processing outlined in an SLA. While, these techniques may be successful in meeting the overall perceived levels of performance when requests being processed over a period of time is considered, none of them can guarantee the same execution times in a consistent manner for every service invocation. Therefore, by design all of them fail to achieve predictable execution times in a repeatable and consistent manner. Such levels of predictability can only be achieved if requests are purposely scheduled to ensure a deadline in a definitive manner. Moreover, the middleware must be designed ground-up with the support required to achieve this level of predictability, from the libraries, development platform and the operating system being used. Additionally, the acceptance of a request for execution must be validated for schedulability ensuring both its deadline and that of the others executing within the same server. With Table 7, some of the related work on Web services are validated against the predictability requirements presented in this paper. As discussed previously, many of them satisfy guideline G4, having some method of service differentiation. Some of them making concious selection of requests for execution based on statistics, partially meet with guideline G3. Conscious scheduling of requests based on a perceived end result partially meets with guideline G2. However, none of them are compliant with guideline G1 nor are they fully compliant with all guidelines identified.

8 Conclusion

The Internet is witnessing a rapid increase in its usage due to service offerings such as Software, Platform and Infrastructure as a service. With Web services playing the role of the facilitator, the performance aspects of Web services demand an increased importance. Among these, execution level predictability becomes far more important due to its impact on the successful delivery of service offerings such as Platform and Infrastructure. Moreover, this would also enable applications with requirements for high predictability in execution to use Web services as a middleware platform, thereby enabling them to reap the inherent benefits of the technology. Through this paper, a set of guidelines, software engineering techniques and algorithms that enable Web services middleware to achieve predictable execution times, were presented. With the two case studies presented, detailed descriptions were given on how the guidelines could be followed in building or modifying existing Web services

middleware products. Moreover, the algorithms and software engineering techniques introduced were generic enough to be used in any middleware product. These were followed by a sample scenario that described how the scheduling and request selection techniques work in unison. The two enhanced Web services middleware products in the case study were evaluated to measure the validity and the performance gain by following the guidelines and the techniques presented. An empirical evaluation was done by comparing the products with their unmodified versions and by comparing the algorithms presented with their non real-time versions. The results obtained clearly prove that predictability of execution can certainly be achieved following the guidelines presented and by using the proper software engineering techniques in these middleware products. Moreover, the results achieved opens up new application areas such as industrial control systems, medical equipment control systems, avionics, robotics and financial trading systems to the use of Web services as middleware.

Future work to be done on this project includes achieving predictability in the network communication aspects. This plays an important role in Web services execution due to its use in highly distributed environments. Although this aspect was considered out of scope for this research, it is vital that predictability in the network layer is also achieved, for this solution to be much more complete. As for the clusters hosting Web services, further research into more specialised dispatching algorithms are already being carried out. With algorithms specialised for specific traffic scenarios and applications, the acceptance rates can further be increased to accommodate the timely execution of more requests. Throughout this paper, some requests found themselves being rejected from execution. Retransmission techniques and rescheduling techniques for such requests are being researched by us to facilitate the execution of these requests without outright rejection. While predictability of service execution can be best achieved with a custom built Web services middleware platform from ground-up, we believe the guidelines and techniques discussed achieve this to a great extent with existing Web services middleware.

Appendix: Sample scenario

The following theoretical example illustrates how the schedulability check and deadline based scheduling work in unison. For brevity, only a single host scenario is considered. While the main difference in the cluster based scenario is the dispatching algorithms used, each of them makes use of the schedulability check internally. Therefore relating this example to the cluster is trivial. Each step identifies the arrival of a request at the system and the schedulability check performed on it. If the request is accepted, the real-time scheduler takes a decision on when the request would be executed. Properties of the requests such as their arrival times, execution requirements and Laxities are summarized in the Table 8, in the order of arrival. The laxity of a request is not used directly in the calculations. However, it is considered (indirectly) when the workload (processor demand) that needs to be completed between the lifetime of a task is considered. The calculations in the example follow Algorithm 1, which is based on the analytical model described in Section 2.

The example starts off with no tasks in the system. The arrival of the first request T1 is shown in Figure 27. As per Table 8, T1 has an execution time requirement of

Table 8 Properties of requests.

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

5 ms that needs to finish within a deadline of 25 ms. In Figure 27, the remaining execution time is illustrated using a dotted line while the deadline has been marked using a straight line. The schedulability check is not carried out for the first arrival.

Request T2 arrives in the system 1 ms later (Figure 28). By the time which, T1 has executed for 1 ms. 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 6–19) is not applicable. However, rest of the check (lines 20–41) is applied as follows.

$$
\begin{aligned}
\text{Proc. demand within} &= (0+6)\ \text{ms} && \text{(Algorithm 1: lines 20–24)}\\
&= 6\ \text{ms}\\
\text{Proc. demand after} &= (0+4)\ \text{ms} && \text{(Algorithm 1: lines 32–36)}\\
&= 4\ \text{ms}\\
\text{Total proc. demand} &= (6+4)\ \text{ms}\\
&= 10\ \text{ms}\\
\text{Loading factor} &= \frac{10}{(25-1)} && \text{(Algorithm 1: line 37)}\\
&= 0.4167\\
0.4167 &> 1 && \text{(Evaluates to } \textit{False} \text{ - accept request)}
\end{aligned}
$$

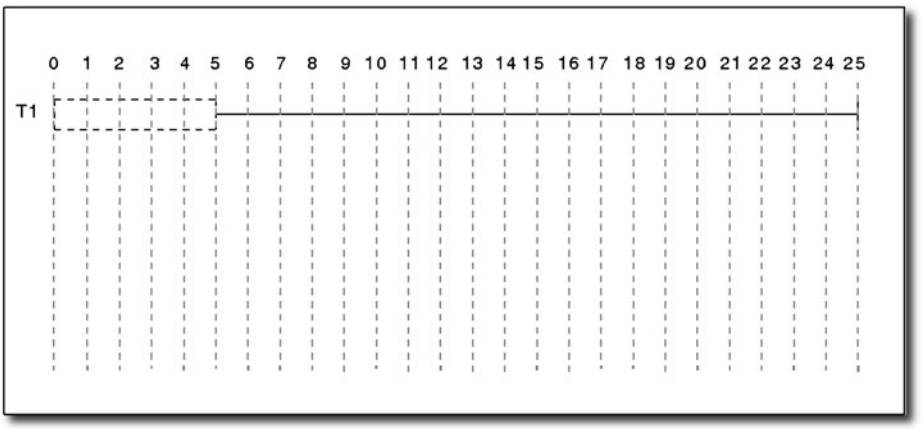

Figure 27 Arrival of request T1.

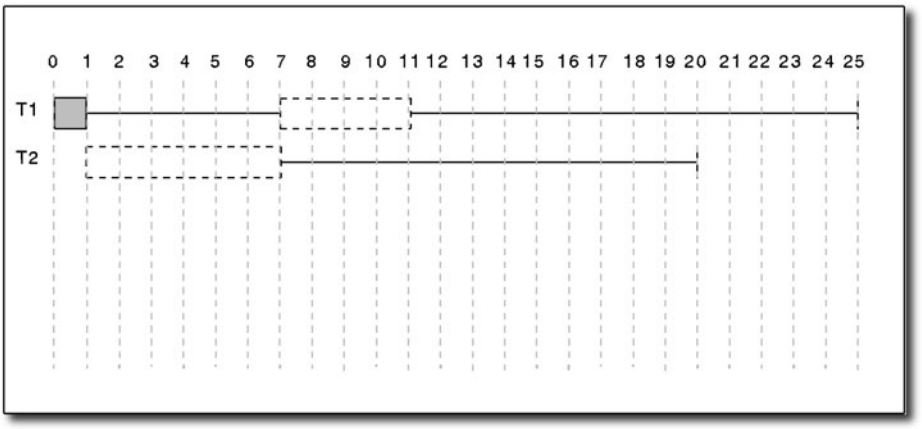

Figure 28 Arrival of request T2.

As visible in the calculation above, the processor demand between the arrival time of T2 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, T1 is preempted and T2 is scheduled for immediate execution by the real-time scheduler. The remaining execution of T1 is delayed till execution of T2 finishes. It can clearly be seen that due to its large laxity, T1 can finish within its deadline. In calculating the loading factor, the total processor demand is divided by the time period between the arrival time of the new task and the deadline of the existing task under consideration (in this case T1). The deadline of the existing task represents its laxity, which indicates the possibility to

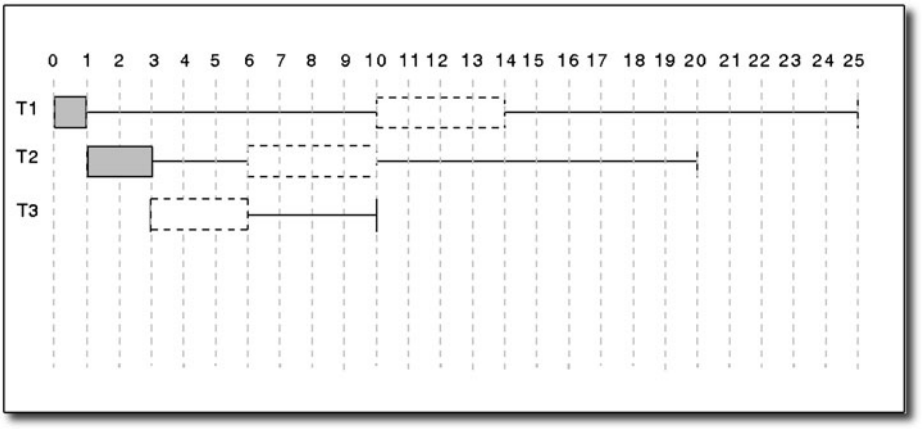

Figure 29 Arrival of request T3.

accommodate the execution of other tasks with deadlines earlier than it, within its lifespan. A higher laxity in the task considered, results in a lower loading factor.

Two milliseconds into the execution of T2, T3 arrives at the system (Figure 29) 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 6–19 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.

$$
\begin{aligned}
\text{Proc. demand within} &= (0+4)\ \text{ms} \\
&= 4\ \text{ms} \\
\text{Proc. demand up to T2} &= (0+4)\ \text{ms} \\
&= 4\ \text{ms} \\
\text{Total proc. demand} &= (4+4)\ \text{ms} \quad \text{(Algorithm 1: lines 32–36)} \\
&= 8\ \text{ms} \\
\text{Loading factor} &= \frac{8}{(20-3)} \quad \text{(Algorithm 1: line 37)} \\
&= 0.47 \\
0.47 &> 1 \quad \text{(Evaluates to } \textit{False} \text{ - continue on to next check)} \\
\text{Proc. demand up to T1} &= (4+4)\ \text{ms} \\
&= 8\ \text{ms} \\
\text{Total proc. demand} &= (4+8)\ \text{ms} \\
&= 12\ \text{ms} \\
\text{Loading factor} &= \frac{12}{(25-3)} \quad \text{(Algorithm 1: line 37)} \\
&= 0.545 \\
0.545 &> 1 \quad \text{(Evaluates to } \textit{False} \text{ - accept request)}
\end{aligned}
$$

The priority queue (RQ) used in Algorithm 1 contains requests in the increasing order of their deadlines. As a result, the check is first performed on the time period between current time and the deadline of T2 and subsequently on the deadline of T1. In calculating the processor demand leading up to the deadline of T2, only the remaining execution time of T2 is considered. T2 having executed for 2 ms, has 4 ms of execution time left. This results in a total processor demand of 8 ms when the execution time requirement of T3 is also considered. When the loading factor is calculated for the time period, its 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 12 ms, 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 29.

With its acceptance, the real-time scheduler preempts T2 by decreasing its thread priority and allows T3 to claim the processor by assigning a higher priority, as it is the task with the earliest deadline. T2 will recommence execution after 4 ms followed by T1 recommencing execution after another 4 ms. Although the execution of T2 is staged and the recommencement of T1 further delayed, the larger laxities of T1 and T2, allow T3 to execute within their lifespans while ensuring all three requests meeting their respective deadlines.

Request T4 arrives at the system 1 ms into the execution of T3 (Figure 30). 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.

$$
\begin{aligned}
\text{Proc. demand within} &= (0+2)\ \text{ms} && \text{(Algorithm 1: lines 6–19)}\\
&= 2\ \text{ms}\\
\text{Total proc. demand up to T4} &= (2+4)\ \text{ms} && \text{(Algorithm 1: lines 20–24)}\\
&= 6\ \text{ms}\\
\text{Loading factor} &= \frac{6}{(11-4)} && \text{(Algorithm 1: line 25)}\\
&= 0.86\\
0.86 &> 1 && \text{(Evaluates to } \textit{False}\text{)}
\end{aligned}
$$

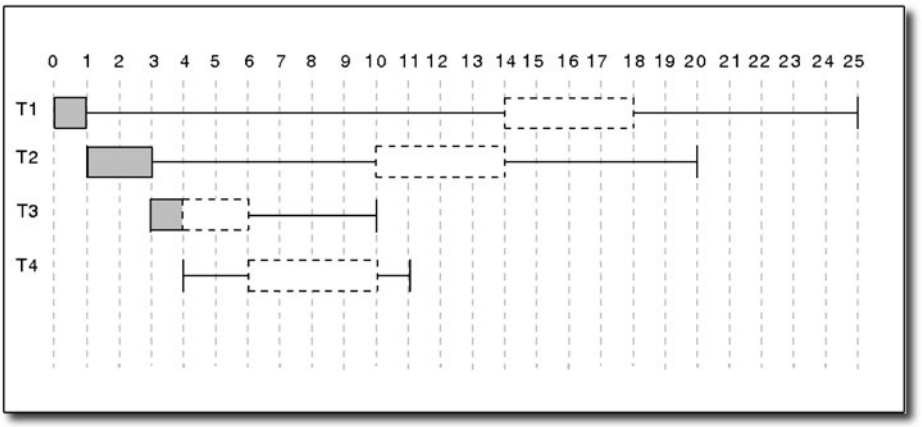

Figure 30 Arrival of request T4.

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

$$
\begin{aligned}
\text{Proc. demand up to T2} &= (0+4)\ \text{ms} \\
&= 4\ \text{ms} \\
\text{Total proc. demand} &= (6+4)\ \text{ms} \quad \text{(Algorithm 1: lines 32–36)} \\
&= 10\ \text{ms} \\
\text{Loading factor} &= \frac{10}{(20-4)} \quad \text{(Algorithm 1: line 37)} \\
&= 0.625 \\
0.47 &> 1 \quad \text{(Evaluates to } \textit{False} \text{ - continue to next check)} \\
\text{Proc. demand up to T1} &= (4+4)\ \text{ms} \\
&= 8\ \text{ms} \\
\text{Total proc. demand} &= (6+8)\ \text{ms} \\
&= 14\ \text{ms} \\
\text{Loading factor} &= \frac{14}{(25-4)} \quad \text{(Algorithm 1: line 37)} \\
&= 0.737 \\
0.737 &> 1 \quad \text{(Evaluates to } \textit{False} \text{ - accept request)}
\end{aligned}
$$

With T4 having a 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 2 ms of execution time of T3 is considered together with the execution time requirement of T4. T4 has a large enough laxity to delay its execution until the remaining execution of T3 is completed within its lifespan. Therefore, the loading factor results to be less than 100% indicating no deadline misses and the check continues to the second part where tasks finishing after T4 is considered.

Processor demand and loading factor for the time period between the deadline of T4 and the deadline of T2 is first carried out. As the loading factor calculates to be less than 100% the same is calculated for the time period between deadline of T4 and the deadline of T1. Both T2 and T1 have large enough laxities to delay their executions further allowing T4 to finish execution within their lifespans, with no deadline misses.

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

T5 is a relatively small task arriving at the system 1ms into the execution of T4 (Figure 31). Moreover, it also has a relatively small laxity. As T5 has a deadline

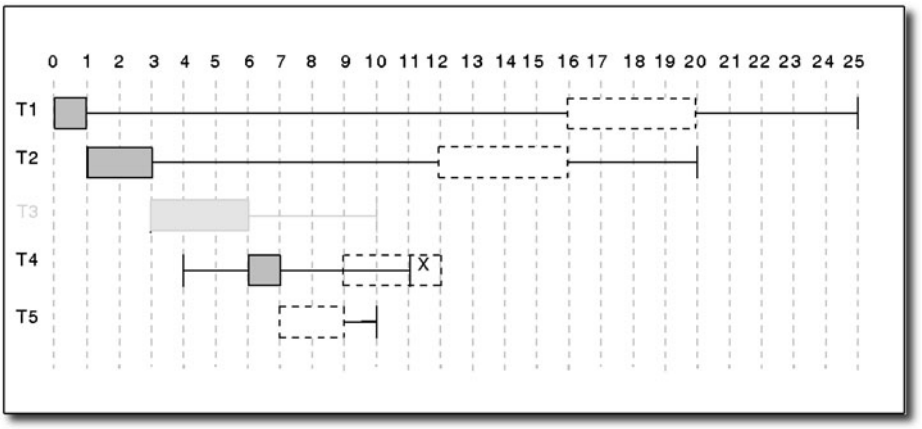

Figure 31 Arrival of request T5.

earlier than the rest of the requests, the first part of the schedulability check is skipped.

$$\begin{aligned}
\text{Proc. demand within} &= (0+2)\ \text{ms} \\
&= 2\ \text{ms} \\
\text{Proc. demand up to T4} &= (0+3)\ \text{ms} \\
&= 3\ \text{ms} \\
\text{Total proc. demand} &= (2+3)\ \text{ms} \quad \text{(Algorithm 1: lines 32–36)} \\
&= 5\ \text{ms} \\
\text{Loading factor} &= \frac{5}{(11-7)} \quad \text{(Algorithm 1: line 37)} \\
&= 1.25 \\
1.25 &> 1 \quad \text{(Evaluates to } \textit{True} \text{ - reject request)}
\end{aligned}$$

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 (3 ms) and the execution time requirement of T5 (2 ms), resulting in 5ms. However, the duration of the time period is just 4 ms (11 ms to 7 ms) in length. As seen above, T4 does not have a large enough laxity to contain the execution of both T5 and its remaining execution time. This results in a loading factor of 1.25 which fails the test. Hence the request T5 has to be rejected.

A loading factor of 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. As T5 would be the task with the earlier deadline, it would gain the CPU continuously till it finishes execution. Thereafter, T4 would be given the CPU as the task with the next earliest deadline. This results in T4 missing its deadline of 7ms from its arrival into

the system, which is the 11th millisecond on the timeline. Rejecting T5 ensures that T4 which is an already accepted request can meet its deadline requirement. After the schedulability with T4 fails, the rest of the schedulability check is skipped.

After the rejection of T5, request T6 arrives at the system 2 ms into the execution of T4 (Figure 32). As T6 has a deadline later than T4, the entire schedulability check is carried out on it.

$$\begin{aligned}
\text{Proc. demand within} &= (0+2)\ \text{ms} \\
&= 2\ \text{ms} \\
\text{Total proc. demand up to T6} &= (2+7)\ \text{ms} \\
&= 9\ \text{ms} \\
\text{Loading factor} &= \frac{9}{(18-8)} \\
&= 0.9 \\
0.9 &> 1 \qquad \text{(Evaluates to } \textit{False}\text{)}
\end{aligned}$$

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

$$\begin{aligned}
\text{Proc. demand up to T2} &= (0+4)\ \text{ms} \\
&= 4\ \text{ms} \\
\text{Total proc. demand} &= (9+4)\ \text{ms} \\
&= 13\ \text{ms} \\
\text{Loading factor} &= \frac{13}{(20-8)} \\
&= 1.083 \\
1.083 &> 1 \qquad \text{(Evaluates to } \textit{True}\text{ - reject request)}
\end{aligned}$$

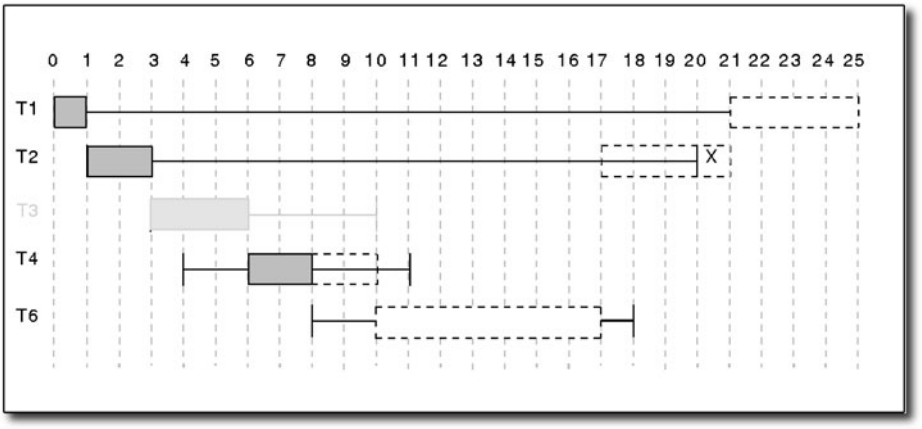

Figure 32 Arrival of request T6.

Unlike T5, T6 having a deadline later than T4 would need to have a laxity that could make way for the remaining execution of T4, without missing its deadline. A loading factor of 90% within the lifespan of T6 means that it could be successfully scheduled to meet its deadline while T4 completes execution within its deadline. In the second part of the schedulability check, the processor demand for the time period between the deadline of T6 and that of T2 is calculated. At this point of time, the laxity of T2 is not adequate to contain the execution of T6 within its lifespan as it already phased its execution making way for T3 and T4. The resultant processor demand and loading factor results in 108% of CPU utilization. This leads to request T6 being rejected.

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

Request T7 arrives at the system, 3ms into the execution of T4 (Figure 33). Having a deadline later than that of T4, the entire schedulability check is applicable.

$$
\begin{aligned}
\text{Proc. demand within} &= (0+1)\ \text{ms} \\
&= 1\ ms \\
\text{Total proc. demand up to T6} &= (1+2)\ \text{ms} \\
&= 3\ \text{ms} \\
\text{Loading factor} &= \frac{3}{(15-9)} \\
&= 0.5 \\
0.5 &> 1 \qquad \text{(Evaluates to } \textit{False}\text{)}
\end{aligned}
$$

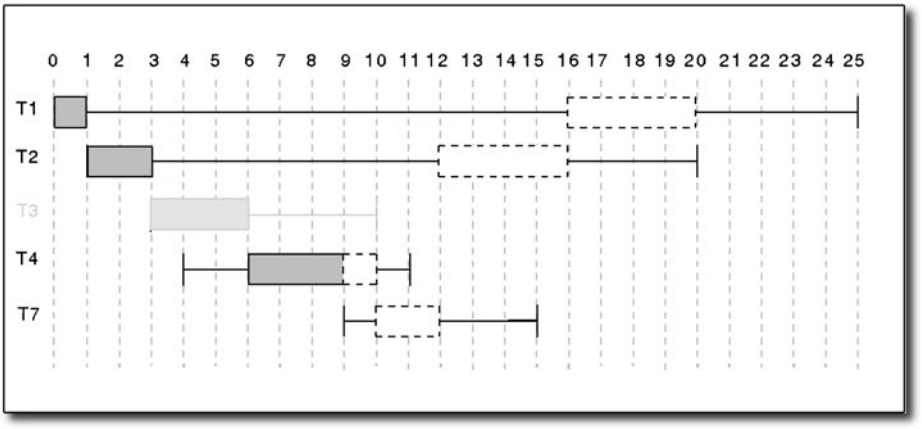

Figure 33 Arrival of request T7.

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

$$\begin{aligned}
\text{Proc. demand up to T2} &= (0+4)\ \text{ms} \\
&= 4\ \text{ms} \\
\text{Total proc. demand} &= (3+4)\ \text{ms} \\
&= 7\ \text{ms} \\
\text{Loading factor} &= \frac{7}{(20-9)} \\
&= 0.636 \\
0.636 &> 1 \qquad \text{(Evaluates to } \textit{False} \text{ - continue to next check)} \\
\text{Proc. demand up to T1} &= (4+4)\ \text{ms} \\
&= 8\ \text{ms} \\
\text{Total proc. demand} &= (7+8)\ \text{ms} \\
&= 15\ \text{ms} \\
\text{Loading factor} &= \frac{15}{(25-9)} \\
&= 0.9375 \\
0.9375 &> 1 \qquad \text{(Evaluates to } \textit{False} \text{ - request accepted)}
\end{aligned}$$

With T4 having only 1ms of execution time remaining, the processor demand between the lifespan of the newly arrived T7 accumulates up to a small 3 ms. T7 has a large enough laxity to comfortably support the execution of T4 and its own within its lifespan. Therefore, the first part of the schedulability check results positive. Next, the schedulability of T7 is checked with T2 and T1. Both these tasks have

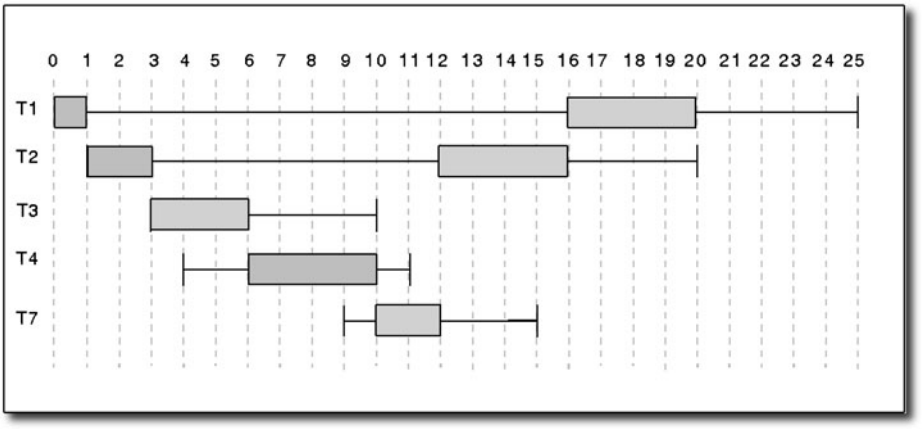

Figure 34 Completed schedule of all accepted requests.

enough laxity to further delay their execution making way for T7 to finish within their lifetimes. The resultant loading factors of 63.6% and 93.75%, indicates that the request T7 can be successfully scheduled with each of the already accepted tasks executing in the system.

Figure 34 illustrates the completed schedule for all accepted requests. Although the execution of some tasks were phased out, all accepted tasks were able to meet their deadlines successfully. Furthermore, it is evident that laxity and the arrival time plays a part in the acceptance of a request. Rejection of requests with deadlines and execution time requirements that cannot be accommodated within the available CPU time, ensures that already accepted requests in the system would not be penalised for their execution.

References

1. Apache Software Foundation: Apache Synapse. http://synapse.apache.org/ (2008). Accessed 16 April 2011
2. Apache Software Foundation: Apache Axis2. http://ws.apache.org/axis2/ (2009). Accessed 16 April 2011
3. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language. Addison-Wesley Professional, Reading (2006)
4. Cao, J., Zhao, H., Li, M., Wang, J.: A dynamically self-configurable service process engine. World Wide Web **13**, 475–495 (2010)
5. Eggert, L., Heidemann, J.: Application-level differentiated services for Web servers. World Wide Web **2**, 133–142 (1999)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Design. Addison-Wesley (1995)
7. Gamini Abhaya, V., Tari, Z., Bertok, P.: Achieving predictability and service differentiation in Web services. In: ICSOC-ServiceWave '09: Proceedings of the 7th International Conference on Service-Oriented Computing, pp. 364–372. Springer, Berlin (2009)
8. Gamini Abhaya, V., Tari, Z., Bertok, P.: Using real-time scheduling principles in Web service clusters to achieve predictability of service execution. In: Proceedings of 8th International Conference on Service-Oriented Computing: ICSOC 2010, San Francisco, CA, USA, pp. 197–212. Springer, Berlin (2010)
9. García, D.F., García, J., Entrialgo, J., García, M., Valledor, P., García, R., Campos, A.M.: A qos control mechanism to provide service differentiation and overload protection to internet scalable servers. IEEE Trans. Serv. Comput. **2**(1), 3–16 (2009)
10. Gartner: SOA is evolving beyond its traditional roots. http://www.gartner.com/it/page.jsp?id=927612 (2009). Accessed 16 April 2011
11. Gartner and Forrester: Use of Web services skyrocketing. http://utilitycomputing.com/news/404.asp (2003). Accessed 16 April 2011
12. Gmach, D., Krompass, S., Scholz, A., Wimmer, M., Kemper, A.: Adaptive quality of service management for enterprise services. ACM Trans. Web (TWEB) **2**(1), 1–46 (2008)
13. Graham, S., Davis, D., Simeonov, S., Daniels, G., Brittenham, P., Nakamura, Y., Fremantle, P., Konig, D., Zentner, C.: Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI, 2nd Edn. Sams Publishing, Indianapolis (2004)
14. Helander, J., Sigurdsson, S.: Self-tuning planned actions time to make real-time SOAP real. In: Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, (ISORC), pp. 80–89 (2005)
15. Kihl, M., Robertson, A., Andersson, M., Wittenmark, B.: Control-theoretic analysis of admission control mechanisms for Web server systems. World Wide Web **11**, 93–116 (2008)
16. Microsoft: Windows communications foundation. http://msdn.microsoft.com/library/ee958158.aspx. Accessed 16 April 2011
17. Pacifici, G., Spreitzer, M., Tantawi, A., Youssef, A.: Performance management for cluster-based Web services. IEEE J. Sel. Areas Commun. **23**(12), 2333–2343 (2005)

18. Ran, S.: A model for Web services discovery with QoS. In: ACM SIGecom Exchanges, pp. 1–10 (2003)
19. Sharma, A., Adarkar, H., Sengupta, S.: Managing QoS through prioritization in Web services. In: Web Information Systems Engineering Workshops, Proceedings, pp. 140–148 (2003)
20. Stankovic, J., Rajkumar, R.: Real-time operating systems. In: Real-Time Systems, pp. 237–253 (2004)
21. Stankovic, J.A., Spuri, M., Ramamritham, K., Buttazzo, G.C.: Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms. Kluwer Academic, Norwell (1998)
22. Sun Microsystems: Glassfish application server—features. http://www.oracle.com/us/products/middleware/application-server/oracle-glassfish-server/index.html (2009). Accessed 16 April 2011
23. Sun Microsystems: Sun Java real-time system. http://java.sun.com/javase/technologies/realtime/ (2009). Accessed 16 April 2011
24. Sun Microsystems: Thread Scheduling Visualizer 2.0—Sun Java RealTime Systems 2.2. http://java.sun.com/javase/technologies/realtime/reference/TSV/JavaRTS-TSV.html (2009). Accessed 16 April 2011
25. Tian, M., Gramm, A., Naumowicz, T., Ritter, H., Freie, J.: A concept for QoS integration in Web services. In: Web Information Systems Engineering Workshops, Proceedings, pp. 149–155 (2003)
26. Tien, C.M., Lee, C.J., Cheng, P.W., Lin, Y.D.: SOAP request scheduling for differentiated quality of service. In: Web Information Systems Engineering—WISE Workshops, pp. 63–72. Springer, Berlin (2005)
27. Wang, A.J., Baglodi, V.: Evaluation of Java virtual machines for real-time applications. Journal of Computing Sciences in Small Colleges **17**(4), 164–178 (2002)
28. Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H., Center, I., Yorktown Heights, N.: QoS-aware middleware for Web services composition. IEEE Trans. Softw. Eng. **30**(5), 311–327 (2004)