Using Real-Time Scheduling Principles in Web Service Clusters to Achieve Predictability of Service Execution

Vidura Gamini Abhaya, Zahir Tari, and Peter Bertok

School of Computer Science and Information Technology
RMIT University, Melbourne, Australia
{vidura.abhaya,zahir.tari,peter.bertok}@rmit.edu.au

Abstract. Real-time scheduling algorithms enable applications to achieve predictability in request execution. This paper proposes several request dispatching algorithms based on real-time scheduling principles that enable clusters hosting web services to achieve predictability in service execution. Dispatching decisions are based on request properties (such as deadline, task size and laxity) and they are scheduled to achieve designated deadlines. All algorithms follow three important steps to achieve a high level of predictability. Firstly, requests are scheduled based on their hard deadlines. Secondly, requests are selected for execution based on their laxity. Thirdly, the underlying software infrastructure provides means of achieving predictability with high precision operations. The algorithms use various techniques to increase the number of deadlines met. One decreases the variance of task sizes at each executor while another increases the variance of laxity at an executor. The algorithms are implemented in a real-life cluster using real-time enabled Apache Synapse as the dispatcher and services hosted in real-time aware Apache Axis2 instances. The algorithms are compared with common algorithms used in clusters such as Round-Robin and Class-based dispatching. The empirical results show the proposed algorithms outperform the others by meeting at least 95% of the deadlines compared to less than 10% by the others.

1 Introduction

With the advent of cloud computing and the trend of exposing 'things' as services on the internet, web services have firmly established itself as the de-facto standard for distributed computing [8,9]. As platforms and infrastructure are being exposed as services, the Quality of Service (QoS) aspects of web service execution mandates an increased importance than before. With applications and systems joining the cloud bandwagon, the performance of web services becomes pivotal to its overlaid applications, thereby requiring stringent QoS levels in their operations. A common practice in alleviating performance bottlenecks, is the use of clusters in hosting web services. The simple idea is to balance the load among service replicas to gain performance. Although a cluster results in a performance gain having multiple hosts and improves the availability of services [18], it does not improve the predictability of web service execution.

Web service middleware are seldom designed to achieve predictability in service execution. Over the years, this has prevented web services being used for applications with critical time requirements and precision in execution. SOAP engines and application

P.P. Maglio et al. (Eds.): ICSOC 2010, LNCS 6470, pp. 197–212, 2010.

servers contain many optimisations for throughput [2,14,16]. For instance, they employ thread pools to service requests in parallel using processor sharing [11]. Although this increases the number of requests being executed within a unit of time, it results in the proportionate increase of the average execution time of a service.

Real-time applications consider predictability of execution with utmost importance. In such systems, missing an execution deadline usually renders even a correctly obtained result useless. As a result, they mandate the use of special scheduling techniques that repeatedly guarantee completion of tasks within requested deadlines. Such stringent requirements in service execution has hindered the use of web services as middleware in real-time systems. In [5,6] we identified three important requirements to achieve predictability of execution. Firstly, the main attribute considered for scheduling a request must be its *deadline* for completion. Secondly, there must be support from the underlying infrastructure (development platform and OS) and thirdly, requests must be selected for execution based on their laxity, which is the indicator of slacktime in a request.

When predictability of execution is considered, previous work on execution time QoS of web services and request dispatching in clusters fail due to many reasons. A common approach for web services is to consider multiple classes of requests [7,15] and provide them with differentiated service. The classes are scheduled to share the processor in a ratio defined by a Service Level Agreement (SLA). They fail due to deadline not being considered as a scheduling parameter. Moreover the infrastructure and software used, do not support predictability. Others [10] use heuristic techniques such as fuzzy logic which cannot guarantee a specific outcome every time. Some [4] support other QoS aspects such as reliability, fault tolerance and does not give prominence to execution. Some of the well known work in request dispatching in distributed systems [3,13,12] cannot be considered as they do scheduling with the goal of reducing the mean waiting time and slowdown of requests, therefore are unable guarantee execution times without specifically scheduling based on deadlines. Neither are they supported by a suitable infrastructure.

In our previous work [5,6] we introduced a solution based on real-time scheduling principles, to achieve predictability in execution on a single server. We selected requests for execution based on their laxity with a guarantee of meeting their deadlines through a schedulability check. The selected requests were pre-emptively executed using earliest deadline first scheduling policy. Furthermore, we presented a real-implementation of the techniques supported by a development platform and an operating system with real-time features. In this paper, we extend our solution to support a cluster.

To address the lack of real-time execution support in web service clusters, our primary contribution through this paper is a set of dispatching algorithms that select and execute requests to meet hard execution deadlines on multiple executors in a cluster. As a secondary contribution, implementation details of these algorithms in a real system, is presented. The uniqueness of our solution lies in the use of real-time scheduling principles and how each of the algorithms function. With RT-RoundRobin we show how a simple algorithm can be made real-time ready by extending it to select requests for execution, based on laxity. The round-robin (RR) nature increases the inter-arrival times at executors which helps them to achieve deadlines. The schedulability check introduced guarantees that a request will not compromise the deadlines of others. RT-ClassBased

is an extension to the popular method of service differentiation through traffic classes. Candidate executors are selected based on the traffic class of a request and the schedulability check tries to place the request among the already accepted, based on its laxity. As traffic classes are based on the execution times, this reduces the variance of task sizes at an executor. RT-LaxityBased makes use of laxities the best possible way. It keeps track of the laxities of requests assigned to each executor and ensures a high variance of laxities at each cluster. As large laxities enable more requests to be scheduled together, spreading them evenly increases the schedulability of requests on the cluster. These three algorithms conduct only a single schedulability check per request on the selected executor. RT-Sequential attempts to achieve a higher schedulability by conducting multiple checks for a single request on different executors. This method makes the best possible use of processing resources, although with a slightly lower deadline achievement rate than the others. All four algorithms make use of our schedulability check from [6] and executors schedule the selected requests using earliest deadline first scheduling principle.

The algorithms are evaluated using a real-life implementation with Apache Synapse [1] used for the dispatcher and Apache Axis2 [2] for cluster servers. We augment the functionality of both these products to be real-time aware in execution. While there are many aspects of QoS in web services, we only consider execution time as as the most important, in this research. Moreover, network communication aspects are considered to be reliable and we make the assumption of communication cost not being significant, in service invocation.

Rest of this paper is organised as follows. In Sect. 2 we provide a background about important task properties for predictable execution. Next we present our solution in Sect. 3 followed by details of the implementation. We present the empirical evaluation in Sect. 5 and discuss some related work in Sect. 6. Finally we conclude in Sect. 7 with a summary and brief look at the way forward.

2 Background

In this section we discuss some important properties of tasks with deadlines to achieve predictability of execution. For a task in execution with a start time S,a deadline D and remaining execution time C, its slack time can be defined as L, where $L = (D - S) - C$. Laxity of a task gives an indication of the same prior to the start of execution, as a ratio between deadline and execution time.

$$Laxity = \frac{Deadline}{ExecutionTime}$$

In the context of deadline based scheduling, laxity and slack time indicates how long a request (or its remaining execution) could be delayed without compromising its deadline. A higher laxity gives ability to delay a task more and schedule other tasks that need to finish earlier. Similarly, a lower laxity means less tasks could be scheduled together.

Figure 2 gives an example on the effect of laxity on scheduling tasks. Tasks T1 and T2 have executed pre-emptively enabling T3, T4 and T5 to be scheduled within their lifespan. It has been possible as a result of T1 and T2 having large laxities. Even with smaller laxities T3 and T4 enable T5, which starts within their lifespan to achieve

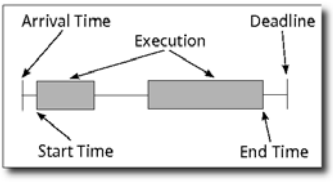

Fig. 1. Properties of a Real-time Task

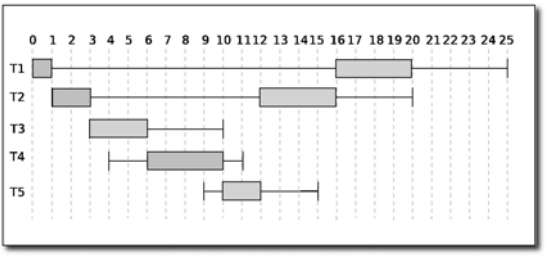

Fig. 2. A deadline based task schedule

its deadline due to its larger laxity. If T5 had a smaller laxity it may not have been schedulable with T3 and T4 as its execution could not have been delayed.

The schedulability check in our previous work [6], selects tasks for execution based on the same principles. Depending on the laxities of tasks it tries to schedule tasks together and ensures that deadline of the target task can be met while not compromising the deadlines of the others. Furthermore, it also prevents a server being overloaded with requests. This check is used by all algorithms presented in this paper for the same purpose and can be identified by the method call *IsSchedulable(newTask, Executor)* where the schedulability of *newTask* is considered with all accepted requests on *Executor*.

3 Real-Time Dispatching Algorithms for Web Service Clusters

The envisioned solution enables a web service cluster to function in a real-time aware manner. Herein, the cluster will honour a hard deadline specified with each request. The deadline for a request is communicated to the cluster using SOAP headers. The solution consists of two components. First, we introduce several real-time aware algorithms to be used at the dispatcher. These would match a request to an executor and ensure the its deadline requirement could be met. The second part of the solution pre-emptively schedules the requests at executors using *Earliest Deadline First* scheduling principle.

3.1 Dispatching Algorithms

The algorithms in our solution perform the task of assigning requests to executors, however with the additional guarantee of meeting the requested deadline. Each of the following algorithms perform in a different way and impacts the variance of laxity at an executor. The goal of the algorithms is to increase the variance of laxities at each executor, thereby increase the number of requests schedulable to meet their deadlines. All algorithms, use our schedulability check presented in [6]. The following algorithms are introduced for request dispatching.

RT-RoundRobin

Round-robin scheduling distributes requests evenly among the executors in a cluster. Adapting this simple yet widely used scheduling technique, RT-RoundRobin (RT-RR)

takes it a step further with an additional schedulability check prior to dispatching a request. Even though RT-RR scheme has little effect on the distribution of request sizes or laxities at an executor, it effectively reduces the arrival rate of tasks. As a result, requests arrive further apart and reduces the number of requests vying to be scheduled within the same window of time. This increases the number of requests schedulable to meet their deadlines.

Algorithm 1 details the steps in RT-RR. The RR nature of it is maintained by keeping track of the last executor a request was assigned to and assigning the new request to the next executor in the list. A check ensures the previous executor assigned is not the last of the list, in which case the list is reset to the beginning (Lines 2-4). Using the index the executor is fetched and schedulability of the request is checked (Lines 7-8). If the request is schedulable, it is assigned to the executor (Lines 9,11) and a reference to the executor is kept as the last one to be assigned (Line 10). A negative result from the check results in the request being rejected (Line 13). Executor information is kept in a data structure with constant time access when the index is used. The schedulability check has a worst case time complexity of $O(n)$ as with the rest of RT-RR. Therefore it results in an overall worst case time complexity of $O(n)$. Moreover, RT-RR is the simplest of the checks with the possible processing overhead kept to a minimum.

RT-ClassBased

Most of the related work on execution level QoS, follow the class based approach where requests are divided into classes based on a priority scheme. These classes get processing time allocations depending on the associated priority in a pre-determined ratio. For instance, consider a system with 3 traffic classes A,B and C. The dispatcher may schedule them based on a 3:2:1 ratio among the executors, where for every 3 requests of class A it schedules 2 requests of class B and 1 of C.

The RT-ClassBased (RT-CB) algorithm follows the same principle with the additional step of ensuring the deadlines of the requests could be met. The class of a request is based on the size of the task identified by the execution time. Executors are mapped with request size ranges, offline and based on it the executor for a given request is selected. With requests assigned to executors based on task size, RT-CB scheduling results in similar sized requests arriving at the same executor. This prevents small and large requests competing for the same executor, thereby increasing the number of small and medium sized requests accepted.

Algorithm 3 contains the steps of RT-CB. The size of the request is used to obtain the executor, through a function that looks-up the mapping information (Lines 1-2). The request is directly checked for schedulability with the executor (Line 3) and assigned to it on being successful (Lines 4-5). On a negative result, the request is rejected (Line 7). Line 1 shows the class of the request being retrieved from itself for brevity. As there is no knowledge of a request prior to its arrival, the size of the request has to be inferred from information at hand. Profiled execution times or execution time history can be used for this purpose. For this, we decided use a combination where profiled times were used when a service was invoked first for a given set of inputs and execution time history was used thereafter. The executor to task size mapping was stored in a data

structure with constant access time. As a result the worst case time complexity of the algorithm is *O(n)* due to the schedulability check.

RT-LaxityBased

The goal of RT-LaxityBased (RT-Lax) is to ensure the even distribution of tasks with higher and lower laxities among the executors. While the schedulability check selects a request for execution based on its laxity and that of already accepted requests, RT-Lax takes it a step further with distributing requests based on laxity prior to using the check. RT-Lax keeps track of the last two laxities assigned to an executor and ensures the same laxities are not assigned to it consecutively. Moreover, it remembers the last executor a request was assigned to and prevents it being considered first for the next request assignment. This increases the variance of laxities at an executor and enables more requests to be scheduled together.

Algorithm 4, describes the steps in the algorithm. Upon receiving a request, the laxity is calculated (Line 1). It is checked to ensure not to be one of the last two laxities assigned to the executor (Line 3-4). In the case of the calculated laxity being in the last two laxities assigned to the last executor, next executor in the list is considered (Lines 13-14). The schedulability check is done on the selected executor (Line 18-19) and the request is either assigned to it or rejected based on the result (Lines 19-24). The first time a request is scheduled through the algorithm, there is no last executor information available. In such a scenario the request is checked for schedulability with the first executor in the list (Lines 30-40). Executor information and details of last laxities assigned to executors are kept in data structures with linear and constant access time complexities respectively. Although a request maybe matched with more than one executor, the schedulability check is conducted only with a single executor. The algorithm bar the schedulability check exhibits a worst case execution time complexity of $O(n)$. Together with the schedulability check, the complete algorithm therefore still results in a time complexity of $O(n)$.

RT-Sequential

RT-Sequential (RT-Seq) algorithm tries to make the best possible match for a request by trying for schedulability more than once. In turn it tries to make best possible use of the server resources by checking the schedulability of a request with more than one executor. If the schedulability check for a request fails with one executor, RT-Seq continues to check its schedulability with the rest of the executors in the cluster until it is schedulable on one of them or the list exhausted. Like RT-Lax this achieves a larger variance of laxity at an executor due to fitting a request ultimately to the best executor. However,it does this with the additional cost of multiple schedulability checks per request. The other algorithms keeps it to a minimum to ensure its cost being too significant, as the lifetime of a request starts from the moment it enters the system.

Algorithm 2 details the steps in RT-Seq. To prevent RT-Seq always starting with the same executor, the successful executor from the last run is kept track of and is considered first (Lines 1,11,22). Requests are repeatedly assigned to it until the check fails (Lines 1-4), in which case another executor is considered (Lines 6-16). Executor information is kept in a data structure with linear access time complexity when accessed sequentially.

The worst case time complexity of the algorithm without the schedulability check is *O(n)*. As multiple schedulability checks may happen for a given request, the overall complexity becomes *O(mn)* where m has an upper bound on the number of executors in the cluster.

Algorithm 1. RT-RoundRobin

```
Require: New request R, List of Executors E, Last Executor
    L
Ensure: R assigned to an executor or rejected
 1. lastExecIndx ← L.getIndex
 2. if lastExecIndx = E.size-1 then
 3.    lastExecIndx = 0
 4. else
 5.    lastExecIndx ← lastExecIndx + 1
 6. end if
 7. nextExec ← E.getExec(lastExecIndx)
 8. S ← IsSchedulable(R,nextExec)
 9. if S = true then
10.    L ← nextExec
11.    Assign R to nextExec
12. else
13.    Reject R
14. end if
```

Algorithm 2. RT-Sequential

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

Algorithm 3. RT-ClassBased

```
Require: New request R, List of Executors E
Ensure: R assigned to an executor rejected
 1. C ← R.getRequestClass
 2. nextExec ← E.GetExecforReqClass(C)
 3. S ← IsSchedulable(nextExec)
 4. if S = true then
 5.    Assign R to nextExecutor
 6. else
 7.    Reject R
 8. end if
```

Algorithm 4. RT-LaxityBased

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

4 Implementation

Dispatcher Component

The algorithms presented were implemented in Synapse using its mediation framework. To support the real-time aspects of the algorithm, all thread-pools in Synapse were replaced with real-time implementations. A real-time scheduler component introduced into Synapse manages the scheduling of the worker threads. Moreover, these additional features were facilitated by running Synapse on Java Real-time System version 2.1 [17] supported by Sun Solaris 10 08/05 real-time operating system (SunOS). The development platform and the operating system provides Synapse with better control, runtime accuracy and precision at the system level which is unavailable for the default implementation.

Executor Component

The executor portion of our solution conducts the important task of scheduling the requests for execution to achieve the deadlines requested. The requests received are scheduled using Earliest Deadline First (EDF) policy in a pre-emptive manner. The real-time enabled Axis2 (RT-Axis2) from our previous work is used at the executors with a few modifications. Unlike in [6], it is relieved of conducting schedulability checks as this is done at the dispatcher. Its functionality is supported by RTSJ and SunOS at the system level.

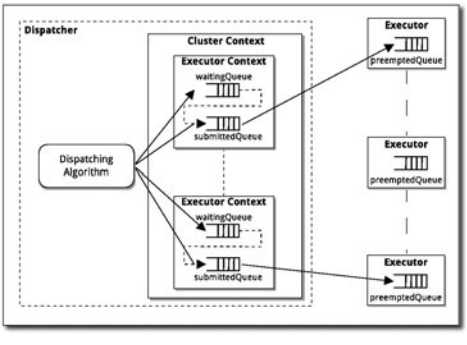

Fig. 3. RT-Synapse Internals

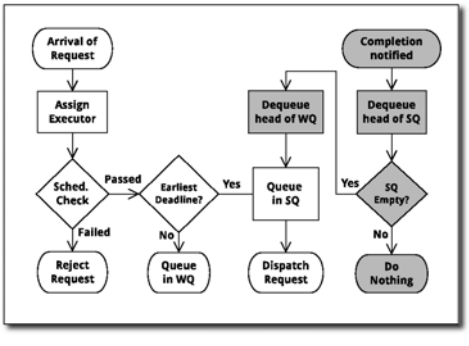

Fig. 4. RT-Synapse Functionality

Cluster Functionality

Figure 3, shows the design of the cluster. Request processing in Synapse is handled by worker-threads and the internal representation of a request is an instance of *RTTask* class. The dispatcher stores the state of the cluster in *Cluster Context* and the state of each executor in instances of *Executor Context* (EC). It keeps track of all tasks assigned to an executor in two ordered queues based on deadlines.

Figure 4 summaries the functionality of the dispatcher. On dispatching a request its RTTask instance is queued on *submittedQueue* (SQ). Requests waiting to be dispatched are queued in *waitingQueue* (WQ). Once a request is accepted for execution, its deadline is compared with already accepted requests. If the new request has the earliest deadline, its RTTask instance is queued in SQ and it is dispatched to the executor. If the new request should not be the earliest to finish, it is queued in WQ. The dispatcher receives a notification from the executor at the completion of a request and its RTTask is removed from the head of SQ. If SQ becomes empty the head of WQ is removed, queued in SQ and the request dispatched to the executor. Conversely if SQ is not empty, requests from the waiting queue are not dispatched to the executor as the executor would be busy processing requests. When the executor is busy, the dispatcher only dispatches a request having an earlier deadline than the one being processed at the executor.

Upon receiving a request, one of two things may happen at the executor. If the executor was idle, the request is taken for processing straight-away. If the executor was busy, deadlines of all requests present is evaluated and the one with the earliest gets the processor to run. The rest are queued on *preemptedQueue* (PQ) which is also ordered on deadlines. At the completion of a request, once the result is returned back to the dispatcher, the head of PQ is removed and its execution resumed. The worker threads that requests are assigned to at arrival, are suspended when a request is pre-empted and woken back up when its execution resumed.

Multi Core/Processor Support

The schedulability check evaluates requests on a single time line of execution. This does not benefit from multi-core and multi-processors hardware. In order to take advantage of such deployments, our solution makes the executors configurable for a number of execution lanes. An execution lane represents a processor core. If an executor is configured for N execution lanes, the same number of requests in the order of their deadlines, would be executed in parallel. On the dispatcher, each lane has its own EC and is considered as a separate executor. As each lane gets its own processor core for execution, no processor sharing takes place among the requests executing in parallel at each executor.

5 Empirical Evaluation

Our implementation is empirically evaluated with traffic generated to resemble various conditions. Our previous work [6] indicated that it is best to test such an implementation with a good mixture of request sizes making the composition of the stream totally random. Hence, for all experiments conducted we use a uniform distribution for task sizes. The deadline for each request is calculated by picking a value between 1.5 and 10 from a uniform distribution and multiplying the profiled execution time for the request. For each run we pick inter-arrival times of requests from a uniform distribution bounded by a low and a high value. We use a web service that allows us to create task sizes with fine-grain precision, using input parameters.

The proposed algorithms select requests for execution based on the schedulability check and those selected are scheduled at the executors using EDF policy.

Table 1. Performance Comparison of Round Robin vs. RT-RoundRobin

	Round Robin (Non real-time)						***RT-RoundRobin***					
	2 Executors		3 Executors		4 Executors		2 Executors		3 Executors		4 Executors	
Inter-arrival time(sec)	% Acc.	% D. Met off % Acc.	% Acc.	% D. Met off % Acc	% Acc.	% D. Met off % Acc	% Acc.	% D. Met off % Acc	% Acc.	% D. Met off % Acc	% Acc.	% D. Met off % Acc
0.25 - 1	99.5	28.8	99.8	37.2	99.9	51.5	88.0	99.0	99.0	100	99.9	100
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 - 0.1	38.8	6.3	52.6	9.1	68.0	13.6	20.5	90.0	37.5	95.0	46.3	99.0

Table 2. Performance Comparisons of Class based vs. RT-ClassBased

	Class based (Non real-time)						***RT-ClassBased***					
	2 Executors		3 Executors		4 Executors		2 Executors		3 Executors		4 Executors	
Inter-arrival time(sec)	% Acc.	% D. Met off % Acc.	% Acc.	% D. Met off % Acc	% Acc.	% D. Met off % Acc	% Acc.	% D. Met off % Acc	% Acc.	% D. Met off % Acc	% Acc.	% D. Met off % Acc
0.25 - 1	100	27.8	99.2	40.8	99.9	58.2	99.2	99.0	100.0	99.0	100	100
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
0.1 - 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

Table 3. Performance of RT-Sequential and RT-LaxityBased

	RT-Sequential						***RT-LaxityBased***					
	2 Executors		3 Executors		4 Executors		2 Executors		3 Executors		4 Executors	
Inter-arrival time(sec)	% Acc.	% D. Met off % Acc.	% Acc.	% D. Met off % Acc	% Acc.	% D. Met off % Acc	% Acc.	% D. Met off % Acc	% Acc.	% D. Met off % Acc	% Acc.	% D. Met off % Acc
0.25 - 1	99.0	96.8	100	97.0	100	97.2	99.2	99.9	100.0	99.9	100	100
0.1 - 0.5	86.0	91.0	96.1	96.3	100	95.0	89.0	99.8	80.5	99.8	99.8	100
0.1 - 0.25	38.6	87.4	76.5	95.0	84.6	96.2	47.4	99.2	66.0	99.6	75.2	100
0.05 - 0.1	29.1	90.0	57.2	95.3	66.7	95.8	38.5	99.0	50.7	99.2	54.3	100

The non-real-time algorithms we compare ours against do not conduct any check for task acceptance. They are implemented directly on non-real-time Synapse and non-real-time Axis2 deployments with best-effort request execution. The three tables listed below summarise the results from different experimental runs. Herein, the success of an algorithm is measured using the percentage of requests accepted (% Acc.) for execution (from the schedulability check) and out of that, the percentage of deadlines met (% D.Met off % Acc.). We conduct all runs with 2 to 4 executors being used in the cluster. % Acc. under non-real-time algorithms used for comparison, signify the percentage accepted due to server overloads resulted by those algorithms.

Round-robin vs. RT-RoundRobin

To illustrate the effect of scheduling based on deadlines, we compare the simple RR algorithm with RT-RR. In RT-RR executors are selected in a RR fashion and the request is checked for schedulability once with the selected executor. Requests that pass the check are accepted for execution on that executor. Table 1 details the results of all runs

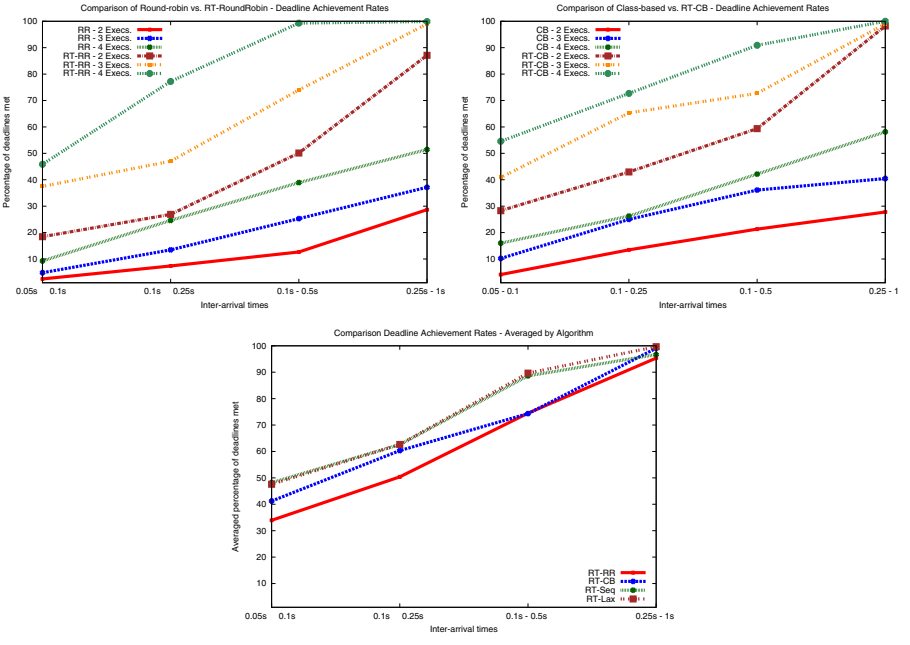

Fig. 5. Execution Time Comparisons

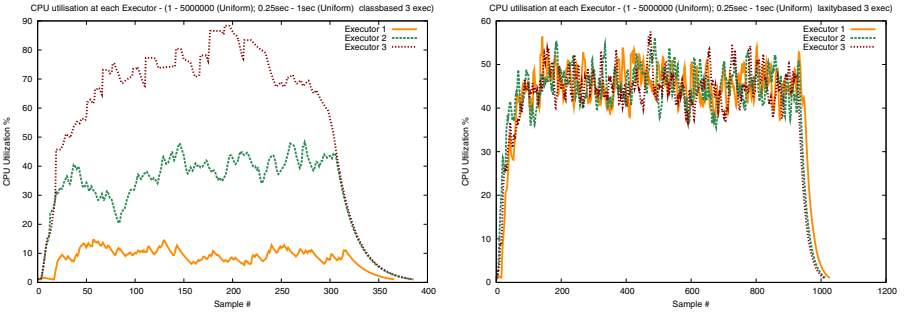

Fig. 6. CPU Utilisation at Executors

conducted. The graph on the left side of the first row in Fig. 5 summarises the deadline acceptance rates. It can be seen that, RT-RR performs better than simple-RR. Simple RR results in better task acceptance rates due to not having an explicit check that rejects request. However, it performs badly with the number of deadlines met. Although RT-RR accepts less requests, it still outperforms simple-RR in meeting more than 90% of the deadlines even in very high request arrivals. With best-effort request execution, the resultant execution times easily miss the deadlines.

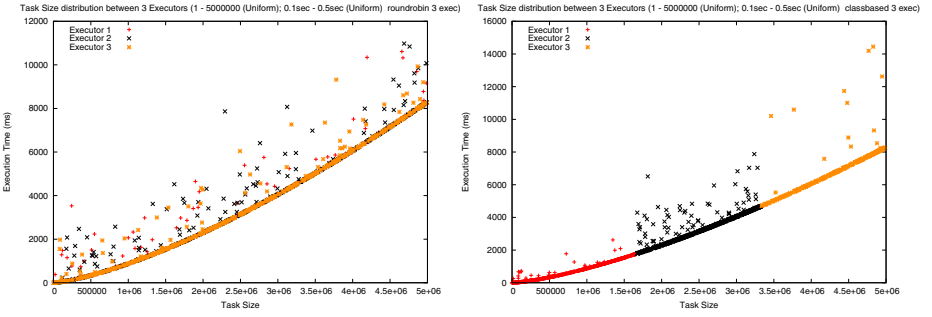

Fig. 7. Task Size distribution at Executors

Class based vs. RT-ClassBased

Next we compare the performance (Table 2) of the class based service differentiation technique with RT-CB, which matches an executor to a request in the same manner, however only accepting the request through a schedulability check. Moreover RT-CB schedules requests for execution based on their deadlines. The graph on the right side of the first row in Fig. 5 summarises the deadline acceptance rates. In both algorithms, requests were sorted into classes based on the size of the request (resultant execution time) and executors were assigned based on the request class. This results in requests of similar sizes being assigned to the same executor and avoids scenarios where a large request and small requests competing for the same server (Fig. 7). As a result, class-based algorithms performs better than simple-RR. However, the percentage of deadlines met are still quite low compared to RT-CB and RT-RR. The first graph on Fig. 6 shows the processor utilisation levels of each executor for an RT-CB runs. The executors exhibit different levels of utilisation due to the classes being based on task sizes.

Performance of RT-Sequential and RT-LaxityBased

RT-Seq checks the schedulability of a request with all executors in the cluster prior to rejecting a request. As a result it returns the highest acceptance rates of all algorithms. However, the total time taken for carrying out multiple checks can become significant for certain requests. Therefore, RT-Seq results in the lowest values for percentage of deadlines met out of all algorithms. These values are still better than the non-real-time algorithms we compare them with. RT-Lax ensures the equal distribution of laxities among cluster members. This creates a higher variance of laxities at each executor thereby enables more requests to be scheduled together. As a result, RT-Lax performs the best in meeting request deadlines. This property of RT-Lax, enables it to make use of the processing resources the best possible way. Compared to a policy such as RT-CB, Fig. 6 shows RT-Lax resulting in equal utilisation levels for all executors in the cluster.

Factors Affecting Request Acceptance and Deadlines Met

From the results presented, it is visible that other factors affect the amount of requests accepted for execution and the deadlines met. As the inter-arrival times of requests decreases, the number of requests accepted and deadlines met also decreases with it. A reduction in inter-arrival times transforms to an increase of requests arriving at the cluster in a given time period. As a result more requests compete for the processor within a single window of time. However, only a portion of them could be accepted for execution. Another factor affecting the acceptance rates and deadlines met is the number of executors in the cluster. Having more executors in the cluster would mean more processing power is available for requests to be executed. Moreover, spreading requests across more executors also results in a reduction of request arrivals at an executor, in a time unit. This enables the executors to successfully achieve the deadlines of requests. Having more executors would mean an increase of workload at the dispatcher. However, when the worst case time complexities of the algorithms are considered, introducing additional executors would not have a big impact on the performance of the dispatcher.

Discussion

The experimental results clearly demonstrate that predictability of execution can be achieved in a web services cluster. In these experiments, we tested the proposed algorithms for the worst possible scenario where there is an equal mixture of task sizes with all requests having hard-deadlines. Moreover, the implementation was exposed to very high request arrival rates. With RT-RR, we demonstrate how a simple scheduling policy could be transformed into being real-time aware. This combines the simplicity of the RR dispatching with the objective scheduling of real-time algorithms. Supported by the underlying software infrastructure RT-RR results in more than 90 % of deadlines being met in all runs. Although simple RR results in higher rates of requests being executed due to its best-effort nature, it fails to guarantee the deadlines of most of them, at times being less than 10%.

With RT-CB we demonstrated how a request-aware dispatching policy could be transformed in being real-time aware. With the additional steps of selecting requests based on their laxity through a schedulability check and deadline based scheduling, it achieves 95% of the deadlines under any traffic condition with more than 50% acceptance rates, at most times. The proposed approach of defining traffic classes based on request sizes and assigning requests for traffic classes, replicate a common tactic used in many other dispatching algorithms. Through this, a request stream consisting of any mixture of requests gets transformed into a request stream with similar sizes at each executor (Fig. 7). This prevents small and large tasks competing for the same executor. This phenomenon, together with decreased arrival rates at executors due to requests being disseminated across many executors enable many requests to be scheduled to meet their requested deadlines. Although the simple class based policy receives the same benefits, the best effort nature of request execution results in deadlines being lost. RT-CB would be suitable for request streams with comparatively more smaller sized requests.

Experimental results confirm that RT-Seq makes the best use of processing resources. It achieved the highest acceptance rates out of all algorithms. Trying to schedule a request repeatedly on different executors ensures that, a request will be scheduled on the

cluster if required processing time is available on any one of the executors. This effectively fills the gaps on processor time lines making the maximum use of their processing resources. However, conducting multiple checks may incur a significant overhead depending on the size of the request. The life of a request, starts on its arrival at the cluster. Therefore, the time spent on being dispatched and being checked for schedulability, has to be subsumed within the execution time requirement of a request. For small sized requests, the overhead incurred by multiple checks may result in them missing their deadlines. As a result, RT-Seq is not suitable for request streams predominantly containing smaller sized requests.

The distribution of requests based on laxity, ensures that requests with large and small laxities are evenly distributed. If an executor gets too many requests with small laxities, eventually some of them will end up being rejected, as they compete for the same window of time. Requests with large laxities are able to shift or stagger their execution within a larger time window enabling more requests to be scheduled within their lifespan. This principle results in RT-Lax meeting the highest number of deadlines, with more than 50% acceptance rate in most cases (second row of Fig. 5). All algorithms show that they could achieve higher performance with the cluster scaling up with more executors. With cost of hardware becoming cheaper by the day, the acceptance rates could be increased with more executors being added to the cluster. In such a setup RT-Lax will be the best algorithm to use with a good mixture of task sizes and laxities in the request stream.

6 Related Work

Much work on dispatching in clusters can be found in the literature. Many of them are for clusters hosting static web content and follow the premise of static web traffic taking a heavy tailed distribution. In [12], no prior knowledge of task sizes are assumed and requests are sent through several executors assigned with increasing quanta until the request is completed in a non-work conserving manner. Requests are mapped to executors based on task sizes in [3,13] and as a result the dispatching transforms the heavy tailed work load into that of type exponential. These work with the goal of reducing the mean waiting time, mean slowdown of tasks and not the predictability of execution. Moreover, with the assumptions they make on static web content such as the heavy-tailed nature of traffic, they seem unsuitable to be used with the highly dynamic nature of web services.

Similarly, evidence of work specific to web service clusters can be found in the literature. In [7] requests are dispatched to achieve probabilistic limits of maximum response time defined in SLAs. A QoS controller monitors the response times resulted in each executor for different types of requests and accepts or rejects requests to maintain the agreed upon limits in the SLAs. While this allows the cluster to achieve a method of controlling response times, the scheduling algorithms used do not consider any deadlines or give a guarantee of achieving them. Requests are divided into traffic classes in [15] and they are scheduled to achieve average response times defined in SLAs. The solution uses a utility function to compute the actual performance of executors and compare them with the rate promised. A discrepancy identified would change the ratio

of the number of requests executed from each class by the executors. Similarly, [10] uses a class based approach where a controller based on fuzzy logic is used to optimise the ratio of requests executed by the cluster.

Commonly, all the work discussed use some aspect related to execution time as a QoS parameter. However, they all fail to guarantee predictable execution times mainly for two reasons. None of them purposely schedule tasks to achieve a deadline in a definite manner. Furthermore, their implementations and infrastructure does not support predictability of execution, by design. Additionally, none of them contain means of validating the schedulability of a request with requests already executing in the cluster.

7 Conclusion

In this paper, we presented four algorithms to achieve predictability of execution in web service clusters. Execution time predictability becomes more important with web services being used to integrate distributed platforms and infrastructure. Moreover, it will enable the use of web services in applications with real-time requirements and enable the real-time development space to enjoy the benefits and advantages web services bring. The proposed algorithms were compared with commonly used request dispatching techniques and experimental results confirm the inability for those techniques to achieve the level of predictability the proposed algorithms demonstrate. We discussed the use of specific algorithms in different types of request streams and their suitability. We also provided brief details about how these algorithms can be implemented in real-life using real-time enabled versions of Apache Synapse and Axis2 for a cluster setup. The software infrastructure used for the implementation also plays a large part in enabling the proposed algorithms to achieve the levels of predictability they demonstrate.

To completely achieve predictability of execution, the network communication aspects of web services would also need to be considered. As future work, we hope to extend our solution to the network level by ensuring the delays are minimised along the path of the web service invocation. We continue to research on more cluster based scheduling algorithms that are more application specific, which would enable them to achieve better acceptance rates.

Acknowledgements. This work is supported by the ARC (Australian Research Council) under the Linkage scheme (No. LP0667600, Titled “An Integrated Infrastructure for Dynamic and Large Scale Supply Chain”).

References

1. Apache Software Foundation: Apache Synapse (June 9, 2008), `http://synapse.apache.org/`
2. Apache Software Foundation: Apache Axis2 (June 8, 2009), `http://ws.apache.org/axis2/`
3. Ciardo, G., Riska, A., Smirni, E.: EquiLoad: a load balancing policy for clustered web servers. Performance Evaluation 46(2-3), 101–124 (2001)

4. Erradi, A., Maheshwari, P.: wsbus: Qos-aware middleware for reliable web services interactions. e-Technology, e-Commerce and e-Service, 2005. In: Proceedings of The 2005 IEEE International Conference on EEE 2005, pp. 634–639 (March April 1, 2005)
5. Gamini Abhaya, V.: Achieving Predictabiliy and Service Differentiation in Web Service Execution. Tech. rep., School of CS and IT, RMIT University, Melbourne, Australia (April 17, 2009), http://goanna.cs.rmit.edu.au/~vabhaya/publications/TechReport1.pdf
6. Gamini Abhaya, V., Tari, Z., Bertok, P.: Achieving Predictability and Service Differentiation in Web Services. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 364–372. Springer, Heidelberg (2009)
7. 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 Transactions on Services Computing 2(1), 3–16 (2009)
8. Gartner : SOA Is Evolving Beyond Its Traditional Roots (April 2, 2009), http://www.gartner.com/it/page.jsp?id=927612
9. Gartner and Forrester: Use of Web services skyrocketing (Septmeber 30, 2003), http://utilitycomputing.com/news/404.asp
10. Gmach, D., Krompass, S., Scholz, A., Wimmer, M., Kemper, A.: Adaptive quality of service management for enterprise services. ACM Transactions on the Web (TWEB) 2(1), 1–46 (2008)
11. 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)
12. Harchol-Balter, M.: Task Assignment with Unknown Duration. Journal of the ACM 49(2), 260–288 (2002)
13. Harchol-Balter, M., Crovella, M., Murta, C.: On Choosing a Task Assignment Policy for a Distributed Server System. Journal of Parallel and Distributed Computing 59(2), 204–228 (1999)
14. Microsoft: Windows Communications Foundation, http://msdn.microsoft.com/library/ee958158.aspx
15. Pacifici, G., Spreitzer, M., Tantawi, A., Youssef, A.: Performance management for cluster-based web services. EEE Journal on Selected Areas in Communications, I 23(12), 2333–2343 (2005)
16. Sun Microsystems: Glassfish Application Server - Features (2009), http://www.oracle.com/us/products/middleware/applicationserver/oracleglassfishserver/index.html
17. Sun Microsystems: Sun Java Real-time System (2009), http://java.sun.com/javase/technologies/realtime/
18. Vilas, J., Arias, J., Vilas, A.: High availability with clusters of web services. Advanced Web Technologies and Applications pp, 644–653