Achieving Predictability and Service Differentiation in Web Services

Vidura Gamini Abhaya, Zahir Tari, and Peter Bertok

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

Abstract. This paper proposes a model and an admission control algorithm for achieving predictability in web services by means of service differentiation. We use real-time scheduling principles typically used offline, adapt them to web services to work online. The proposed model and algorithm is empirically evaluated by implementing it Apache Axis2. The implementation is benchmarked against the unmodified version of Axis2 for various types of workloads and arrival rates, given different deadlines. We meet 100% of the deadlines keeping a healthy request acceptance rate of 42-100% depending on the task size variation. Our solution outperforms Axis2, specially at instances with high task size variance, by a factor of 10 - 1000.

1 Introduction

Web service architectures and supporting infrastructure (such as SOAP engines and application servers) by design, lacks support for predictability in execution. For instance, they service requests in a *best effort* manner. As a result, specialised middleware (such as Real-Time CORBA [1]) has been the default choice for applications with real-time requirements.

Applications with real-time requirements are characterized by the equal importance placed on time taken for a result to be obtained as on the correctness of the computation performed. Herein, the notion of time taken for the result to be obtained is expected to be predictable and consistent invariably. Moreover, if the time taken to obtain the result is beyond a certain deadline, the result may be considered useless and might lead to severe consequences. As a result, real-time systems with stringent QoS levels require the service execution and middleware used to have very high predictability [2].

Although some research in web services has attempted to address Quality of Service (QoS) aspects [3,4,5,6,7], none of them guarantees predictability in all aspects of functionality, such as message processing and service execution. Moreover, there is no support for predictability from the operating system (OS) and the development platform. The work that comes close to achieving it [8], does it in a confined embedded environment where tasks and their resource requests are known in advance of the task occurrence. The challenge would be to achieve predictability in the totally dynamic environment that web services are used in.

Contribution. Our solution is unique due to several reasons. The solution achieves predictability in a highly dynamic environment with no prior knowledge of requests.

L. Baresi, C.-H. Chi, and J. Suzuki (Eds.): ICSOC-ServiceWave 2009, LNCS 5900, pp. 364–372, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Request acceptance is not pre-determined and happens *on-the-fly*. Moreover, it allows any web service request to be tagged with a target completion time, which on acceptance is guaranteed to be met. This is the first approach of scheduling web service requests based on a user requested deadline. Our solution is unique in adapting real-time scheduling principles designed for offline use, to a dynamic online environment.

2 Background

Two concepts considered in schedulability analysis, namely *processor demand* and *loading factor* [9] are defined here. Henceforth, we use a given task T_i, with release time of r_i, a deadline of d_i and an execution time requirement of C_i. Our proposed model is based on the following definitions.

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

$$h_{[t_1,t_2)} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k. \tag{1}$$

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

$$u_{[t_1,t_2)} = \frac{h_{[t_1,t_2)}}{t_2 - t_1}. \tag{2}$$

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

$$u = \sup_{0 \leq t_1 \leq t_2} u_{[t_1,t_2)}. \tag{3}$$

Theorem 1 (Spuri [10]). *Any set of real-time tasks is feasibly scheduled by EDF algorithm if and* only *if*

$$u \leq 1. \tag{4}$$

3 Proposed Solution

3.1 Proposed Model

The proposed model is based on the notion of a *deadline*, specified by the client at the time of service invocation. The Deadline is considered to be the absolute time period the request has to be serviced within. The proposed solution has two parts. An *on-the-fly* schedulability check is conducted on the arrival of a task at the system, to evaluate the possibility of servicing it within the requested deadline, without compromising already accepted tasks. Then the accepted tasks are scheduled using a deadline based real-time scheduling algorithm.

In a pre-emptive scheduling system, execution of a given request could happen with several pre-emption cycles.

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

$$E_i = \sum_{j=1}^{n} (e_j - s_j). \tag{5}$$

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

$$R_i = C_i - E_i. \tag{6}$$

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 definition 1, the processor demand within the duration of the newly submitted task can be defined as,

$$h_{[r_{new},d_{new})} = \sum_{r_{new} \le d_p \le d_{new}} R_p + C_{new}. \tag{7}$$

With reference to definition 2, the loading factor within the duration of the newly submitted task can be defined as,

$$u_{[r_{new},d_{new})} = \frac{h_{[r_{new},d_{new})}}{d_{new} - r_{new}} \tag{8}$$

With condition 8, if the following condition is satisfied, the new task is considered schedulable together with tasks finishing on or before its deadline, with no impact on their deadlines.

$$u_{[r_{new},d_{new})} \le 1 \tag{9}$$

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

$$h_{[r_{new},d_q)} = h_{[r_{new},d_{new})} + \sum_{d_{new} \le d_i \le d_q} R_i. \tag{10}$$

The result of 7 is used as part of the equation. This represents the processor demand of all tasks finishing on or prior to d_{new} and can be treated as one big task with a release time r_{new} and a deadline of d_{new} respectively. Next, the loading factor for the same duration is calculated.

$$u_{[r_{new},d_q)} = \frac{h_{[r_{new},d_q)}}{d_q - r_{new}} \tag{11}$$

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

$$u_{[r_{new},d_q)} \leq 1 \tag{12}$$

In summary, for a newly submitted task to be accepted to the system, condition 9 needs to be satisfied for tasks with deadlines on or before d_{new}, subsequently condition 12 needs to be satisfied, separately for each task with deadlines after d_{new}.

3.2 Proposed Algorithm

Based on the above model, Algorithm 1, will form the core of our solution in the implementation that follows. In devising the algorithm, we make the assumption that the execution time requirement or an estimation of it per parameter, for each service hosted would be available to the server.

Current time, deadline of the new request and the list of requests already accepted by the system as inputs. Current time is considered as the start time of the new request. As per the model described in 3.1, The check consists of two steps. First part determines the schedulability of a new request together with tasks finishing within its lifespan, while meeting all deadlines (Lines 2 to 14). For each request $P' \in P$, it is checked whether execution information is currently available (Line 4). If the request has been partially processed, the remaining execution time calculated as per equation 5, is obtained (Line 5). If the request is yet to be processed, the execution time requirement of the task (Line 7), is used alternatively.

Following equation 7, the processor demand within the duration of the newly submitted request is calculated by summing up the remaining execution times or execution time requirements of each task. Adding the execution time requirement for the new request (Line 10) completes the processor demand calculation for its lifespan. Following equation 8, we calculate the loading factor for the time period (Line 11). If the loading factor is greater than 1, the request is straightaway rejected. If the loading factor remains less than 100%, the check continues on to the second stage.

The second stage validates the effect of the newly submitted request on requests finishing thereafter. For this we select requests with deadlines later than that of the new task (Line 15). The check is done separately for each and every request selected. We make the process more efficient by, first sorting the list of selected requests in the ascending order of the deadlines (Line 16). For each request $Q' \in Q$, a further subset of requests from the list is selected. All requests required to finish between newly submitted and Q' are selected into set R (Line 19). For each request $R' \in R$, the processor demand is calculated by using either the remaining time of the request or its execution time requirement (Lines 21 to 26). To this, the remaining time or the execution time request of request Q' is also added (Line 27 to 29). Following equation 10, the processor demand calculated for the duration of the new request is added to it (Line 28 and 30). Finally, following equation 11, we calculate the loading factor for the same duration (Line 32). If the result exceeds 100%, the request is rejected (Line 33 to 35). If it is less than or equal to 100%, the check is repeated for members in Q, until a check fails or all of them are satisfied, at which point the request is considered schedulable.

Algorithm 1. Online Schedulability Check

Require: (S_{new}) Current time, (D_{new}) deadline of new request *N*, (*T*) List of requests currently in the system
Ensure: true: if the task can be scheduled, false: if the schedulability check fails

1: WPD ← 0; APD ← 0
2: P ← GetTasksFinWitnNewTask(T, S_{new} , D_{new})
3: **for all** P′ ∈ P **do**
4: **if** execution information for P′ exists **then**
5: WPD ← WPD + GetRemExTm(P′)
6: **else**
7: WPD ← WPD + GetExecTime(P′)
8: **end if**
9: **end for**
10: WPD ← WPD + GetExecTime(N)
11: LoadingFactor ← $\frac{WPD}{(D_{new}-S_{new})}$
12: **if** LoadingFactor > 1 **then**
13: **return** false
14: **end if**
15: Q ← GetTasksFinAftNewTask(T,D_{new})
16: Q ← SortByDL(Q)
17: **for all** Q′ ∈ Q **do**
18: DL ← GetDL(Q′)
19: R ← GetTasksFinBtwn(T, D_{new}, DL)
20: **for all** R′ ∈ R **do**
21: **if** execution information for R′ exists **then**
22: APD ← APD + GetRemExTm(R′)
23: **else**
24: APD ← APD + GetExecTime(R′)
25: **end if**
26: **end for**
27: **if** execution information for Q′ exists **then**
28: APD ← APD + WPD + GetRemExTm(Q′)
29: **else**
30: APD ← APD + WPD + GetExecTime(Q′)
31: **end if**
32: LoadingFactor ← $\frac{APD}{(DL-S_{new})}$
33: **if** LoadingFactor > 1 **then**
34: **return** false
35: **end if**
36: **end for**
37: **return** true

Sorting the requests by ascending deadlines ensures a failure happens as early as possible. Moreover, it avoids the check being repeated after a failure. The complexity of the algorithm results in $O(n^2)$.

3.3 Deadline Based scheduling

The requests accepted through the schedulability check are scheduled using a deadline based scheduling algorithm. It schedules tasks sequentially in the increasing order of their deadlines. The algorithm makes use of priority levels to control the execution of the worker threads in the system. This ensures predictability at execution level of the system.

4 Empirical Evaluation

The implementation is benchmarked against unmodified version of Apache Axis2. Since Axis2 by design works in a best effort manner, there would be no resultant task rejections. However, the number of tasks meeting their deadlines is used as the main metric to measure performance. A web service that allowed us to fine tune the task sizes with a single parameter was used for the experiments. We generated task sizes according to Uniform, Exponential and Pareto distributions. Moreover, we generated task arrival rates using a Uniform distribution.

4.1 Experimental Results

The success of our solution, depends on two primary factors. The number of requests accepted for execution and the number of requests meeting the requested deadlines. The aim was to achieve a high rate of task acceptance and to ensure that majority of them met their deadlines. The task size distribution, the execution time to deadline ratio and the arrival rates had an effect on this. Table 1 contains a summary of all experiments runs. The first two columns contain the various experiment runs conducted and the parameters used. The deadline for each run was calculated by multiplying the respective profiled execution time requirement by a value between 1.5 and 10 drawn out uniformly from the distribution.

Table 1. Comparison of Real-time Axis2 and Unmodified Axis2 performance

Distribution	Inter-arrival time(sec)	*Real-time Axis2* % Acc.	% D. Mis.	% D. Met	*Unmodified Axis2* % D. Mis.	% D. Met
Uniform	0.25 - 5	41.8	0	100	96.6	3.4
	0.25 - 10	81.2	0	100	83.6	16.4
Bounded Exponential $\lambda = 10^{-6}$	0.25 - 2	62.5	0.1	99.9	42.6	57.4
	0.25 - 5	99.3	0	100	0	100
	0.25 - 10	100	0	100	0	100
Bounded Exponential $\lambda = 10^{-5}$	0.25 - 2	100	0	100	0	100
	0.25 - 5	100	0	100	0	100
	0.25 - 10	100	0	100	0	100
Bounded Pareto $\alpha = 0.5$	0.25 - 2	100	0.3	99.7	0	100
	0.25 - 5	100	0.1	99.9	0	100
	0.25 - 10	100	0	100	0	100
Bounded Pareto $\alpha = 0.05$	0.25 - 2	99.4	0	100	0	100
	0.25 - 5	99.9	0	100	0	100
	0.25 - 10	100	0	100	0	100

4.2 Discussion

Task Acceptance. In a given period of time, real-time Axis2 accepts between 42% - 100% tasks it receives, depending on the mixture of tasks. If the request sizes take an Exponential or a Pareto type distribution, the task acceptance rate results a 100% in almost all the experiment runs. This happens due to the high concentration of small sized requests in the mix. A small task may have an execution time requirement of a few CPU cycles or even a fraction of a CPU cycle. As a result, it is possible to finish more small requests in a given period of time. Medium and large tasks having a higher execution time requirement tend to get accumulated to the backlog of tasks waiting to finish execution. This results in task rejections.

Impact of Arrival Rate. Varying the arrival rates of requests, it was observed that the number of tasks accepted is proportional to the arrival rate of tasks. A higher arrival rate results in the system receiving requests at a higher rate than it is completing requests. This leads to a build-up of unfinished requests in the system that leads to the eventual rejection of tasks. Moreover, a lower arrival rate results in the system finishing up execution of requests before the next task arrives at the system. This yields a higher task acceptance rate.

Deadline Achievement. With the real-time implementation, 100% of the requests achieve their deadlines, in most cases. It is clearly visible that the real-time implementation performs better than unmodified Axis2 in meeting request deadlines. When the requests received are predominantly small, both versions of Axis2 performs well with meeting deadlines of all tasks. Due to the small execution time requirements of the tasks the deadlines are easily achieved as it leads to no task build-up. Unmodified Axis2 performs marginally better than the real-time implementation in a couple of cases were the task mix consist of only very small sized requests.

Timeliness of execution. According to Fig. 1, with runs having a higher variety of task sizes, real-time Axis2 results in better execution times than unmodified Axis2 by very large factors. When the requests are predominantly small, both implementations achieve

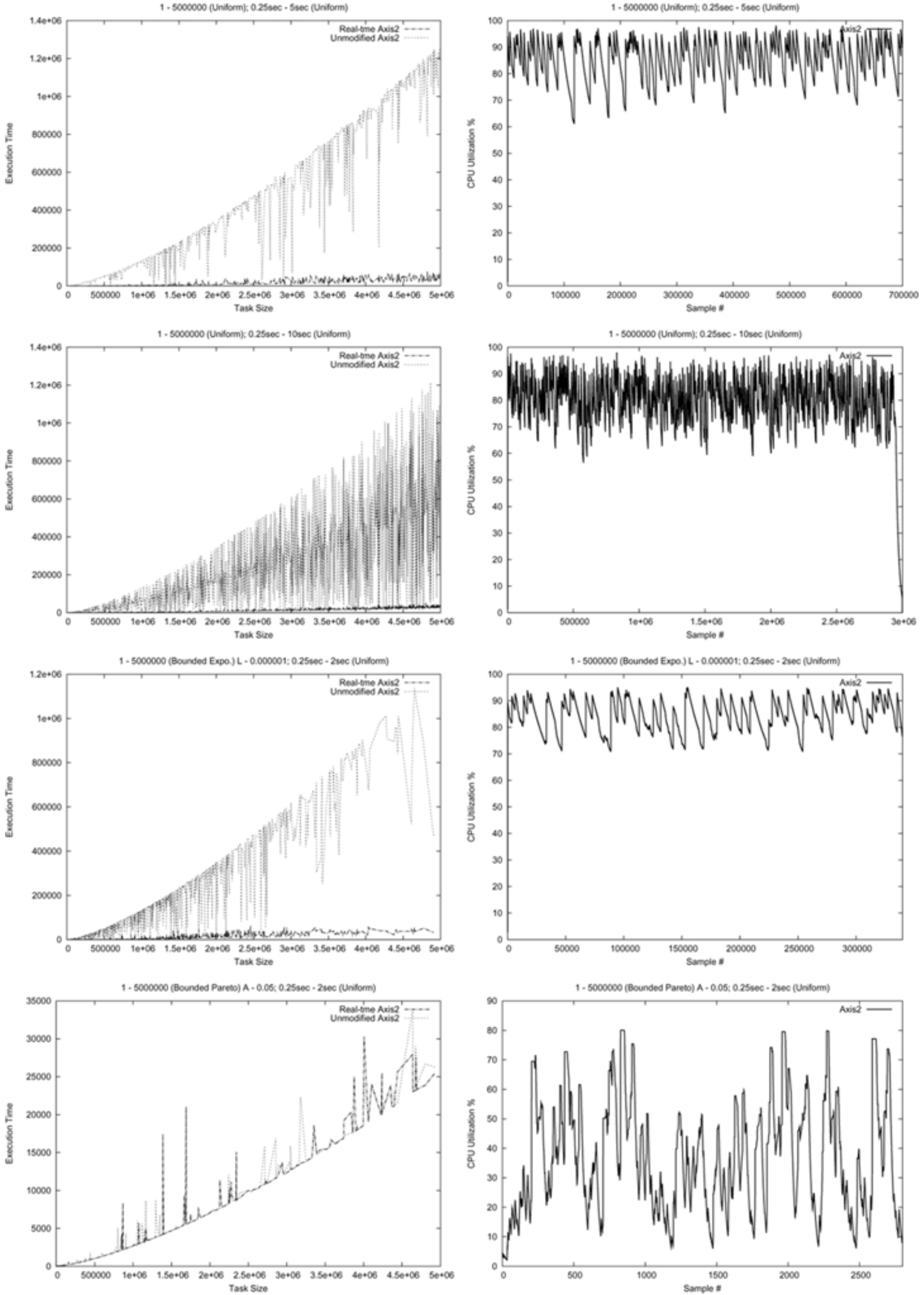

Fig. 1. Execution Time Comparison of Unmodified and Real-time Axis2 implementations

deadlines of almost all the requests. This is largely due to the best effort nature that unmodified Axis2 functions in. Accepting all requests and trying to execute as many as possible in parallel, results in all requests taking a longer time to finish execution. With the real-time implementation, a task is only accepted if its deadline could be met while not compromising that of the others already accepted. This results in lower execution times compared to unmodified Axis2. In some of the bounded exponential and pareto runs, Real-time Axis2 execution times have resulted in higher than normal execution times for certain task sizes. These are deviations intended behaviour by the real-time scheduler, in order to achieve the deadlines of other tasks.

CPU utilisation. In the runs where there was rejection of tasks, it is clearly visible from Fig. 1 that Real-time Axis2 has a very high utilization of the process during the experimental runs. Although it is reasonable to assume that the processor should be utilized 100% of the time by a process in such scenarios, practically this may not achieved due to thread level scheduling. However, with a real-time OS being used and a development platform that supports real-time systems, it is clearly visible that very high rates of processor utilisation can be achieved. Moreover, whenever tasks are rejected by the schedulability check, it is backed up by the high processor utilisation.

5 Conclusion

With the experiment results discussed in the previous section, it could be concluded that the devised model, schedulability check introduced and the real-time algorithm achieved their purpose. Moreover, it would be fair to conclude that real-time Axis2 achieves the goal of maximizing request deadline achievement. The result show a significant difference in the execution times achieved especially where there is a high variety of task sizes. The solution performs decently with a task acceptance rate of 42 - 100% varying with the arrival rates of tasks. Most importantly it performs well to achieve the deadline of all accepted tasks consistently.

References

1. Schmidt, D., Kuhns, F.: An overview of the Real-Time CORBA specification. Computer 33(6), 56–63 (2000)
2. Schmidt, D., Levine, D., Mungee, S.: The design and performance of real-time object request brokers. Computer Communications 21(4), 294–324 (1998)
3. Ran, S.: A model for web services discovery with QoS. ACM SIGecom Exchanges 4(1), 1–10 (2003)
4. 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)
5. Yu, T., Lin, K.: The design of QoS broker algorithms for QoS-capable web services. In: IEEE International Conference on e-technology, e-commerce and e-service, EEE 2004, pp. 17–24 (2004)
6. Sharma, A., Adarkar, H., Sengupta, S.: Managing QoS through prioritization in web services. In: Web Information Systems Engineering Workshops, Proceedings, December 2003, pp. 140–148 (2003)

7. Tien, C.-M., Cho-Jun Lee, P.: SOAP Request Scheduling for Differentiated Quality of Service. In: Dean, M., Guo, Y., Jun, W., Kaschek, R., Krishnaswamy, S., Pan, Z., Sheng, Q.Z. (eds.) WISE 2005 Workshops. LNCS, vol. 3807, pp. 63–72. Springer, Heidelberg (2005)
8. 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)
9. Stankovic, J.A., Spuri, M., Ramamritham, K., Buttazzo, G.C.: Deadline scheduling for real-time systems: EDF and related algorithms. Kluwer Academic Publishers, Dordrecht (1998)
10. Spuri, M.: Earliest Deadline scheduling in real-time systems. Doctorate Dissertation, Scuola Superiore S. Anna, Pisa (1995)