TRANSPARENT SCHEDULING OF WEB SERVICES

Dmytro Dyachuk and Ralph Deters
Department of Computer Science, University of Saskatchewan, 110 Science Place, Saskatoon, Canada

Keywords: SOA, Web Services, Scheduling.

Abstract: Web Services are applications that expose functionality to consumers via public interfaces. Since these interfaces are defined, described and consumed using XML-based standards, Web Services outperform other middleware approaches (e.g. CORBA, RPC) in terms of platform interoperability and ease of use. Web Services support the concept of loosely coupled components, which in turn enables the development of more agile and open systems. However, this flexibility comes at the price of reduced control over the usage of the services that are exposed via the interfaces. This paper focuses on the transparent scheduling of inbound requests by introducing a proxy that prevents clients from directly accessing the provider. By manipulating the order and volume of requests sent to the provider it becomes possible to improve throughput and mean response time and to ensure consistent performance in overload situation.

1 INTRODUCTION

According to the *Four Tenets of Service Orientation* (Box 2004), services are characterized by having *clear boundaries* and *autonomy*. Service Orientation (SO) also replaces data types as a means to describe input/output of services by using *contracts and schemas* and defines service compatibility by use of *policies*.

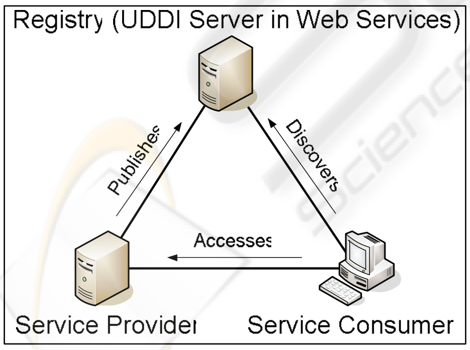

Figure 1: Service-Oriented Architecture.

In a service-oriented system (fig. 1), services are offered by service providers that register them with registries (e.g. UDDI). Service consumers (aka clients) discover at runtime service providers by simply queering the registries. Upon discovering a service provider, the consumer obtains from the provider the meta-data of the service that is then used to establish a binding to the provider. Since services are high-level constructs that hide implementation details, consumers can easily bind to unknown services across platform and language barriers, resulting in a system with very dynamic functional dependencies between its components. Consequently SO supports very loose coupling between consumers and providers, allowing for agile and open systems. Compared to other middleware approaches such as RPC (e.g. ONC-RPC) and object-oriented middleware (e.g. CORBA) SO differs in its lack of access transparency, since there is a very clear notion between local and remote.

Service-oriented middleware (e.g. Web Services) enables developers to expose functionality in terms of services, which can be described in a declarative manner ensuring interoperable and platform independence. Using IDE tools (e.g. Visual Studio 2005, Eclipse 2006) and frameworks (e.g. Axis 2006) it is fairly easy for programmers to expose application interfaces and/or consume existing services, resulting in an ever-increasing number of Web Service deployments.

However, the ease with which components (e.g. legacy systems) can now be exposed and consequently combined, raises serious concerns in regards to the dependability of the resulting system.

It is important to remember that providers of services expose local resources (e.g. legacy applications) to potentially new and unknown loads.

This is particularly worrisome since Web Services platforms tend to implement PS (Processor Sharing) as the default scheduling policy (Graham 2004) which can easily lead to server overload situations that in turn can cause ripple effects throughout a system (e.g. fault-propagation).

This paper focuses on the use of scheduling as a means to ensure that the exposed services are protected from overload situations and that the throughput and mean response time are optimized. The remainder of the paper is structured as follows. Section two presents an overview on server behaviour and scheduling. Section three presents a benchmark and an experimental setup of a system used to evaluate the scheduling of service requests. This is followed by experimentation and evaluation sections that discuss the results of the experimentation. Section six discusses related work. The paper concludes with a summary and outlook.

2 SERVERS & LOAD

If we assume that service providers do not share resources with other service providers (e.g. no two providers expose the same data base), than every service provider can be modelled as a server.

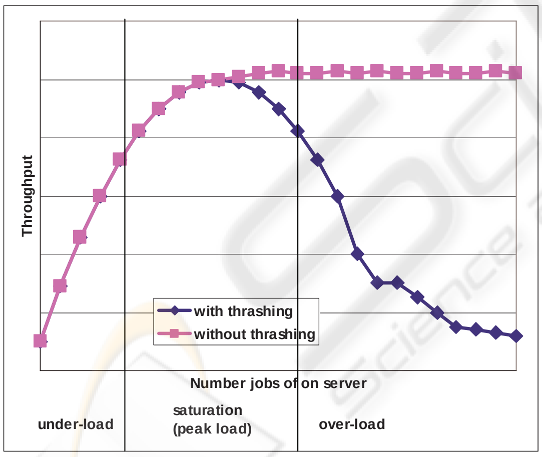

Figure 2: Behaviour of Server.

If these providers are capable of handling multiple requests simultaneously, incoming service requests must be assigned a separate thread. Since each server has a finite amount of resources and each consumer request will lead to a temporary reduction of server-resources, it is interesting to examine the server's behaviour under various loads. Studies (Heiss 1991) show that servers respond in a common way to loads. If a server is gradually exposed to an ever-increasing number of service requests it is possible to observe three distinct stages, *under-load*, *saturation* and *over-load*. At the beginning the server experiences a load that is below its capacity (*under-load*) and consequently it is not fully utilized. As the number of requests is increased the throughput (number of completed jobs per time unit) increased. As the rate of incoming requests increases the server experiences its *saturation* point (peak load). This saturation marks the point where the server is fully utilized and operating at its full capacity. The saturation point marks also the highest possible throughput. Further increases of the rate at which the request arrive will now lead to an overload or *thrashing effect*. The service capacity is exceeded and "an increase of the load results in the decrease of throughput" (Heiss 1991). The main reasons for a server to experience thrashing are either *resource* contention (overload of the physical devices e.g. CPU, memory, hard drive, etc.) or *data* contention (locking).

Since server over-load situations lead to a decline in throughput it is important to avoid them. Heiss and Wagner (Heiss 1991) proposed the use of adaptive load control as the means for preventing overloads. This is achieved by first determining the maximum number of parallel requests (e.g. maximum number of simultaneous consumers) and then buffering/balking the requests once the saturation point has been reached.
The impact of this approach can be seen in figure 2. The darker curve shows the characteristic three phases a server can experience, under-load, saturation and over load. Using an admission control (grey curve), the trashing is avoided due to the buffering/balking of requests above peak load.
Adding an admission control into an already existing Web Services system can be achieved by use of the proxy pattern. As shown in figure 3, adding a proxy that shields/hides the original provider enables a transparent admission control.

The role of the proxy is to monitor the rate at which consumers issue requests. Once the request rate exceeds the capacity of the provider, a FIFO queue is used to buffer the excess request. The transparent admission control is a very effective approach for handling requests bursts (Heiss 1991, Elnikety 2004). However a basic FIFO queue ignores that different requests impact the server in different way e.g. different resource utilization.

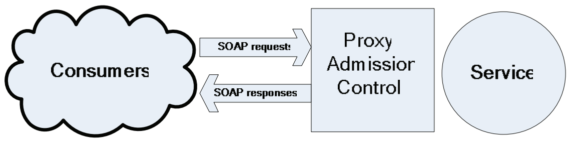

Figure 3: Transparent Admission Control.

Since service-oriented middleware (e.g. Web Services) tends to favour declarative communication styles (e.g. SOAP messages), it is fairly easy to analyze the server bound traffic, determine the request and estimate the impact each request will have on the service provider. This in turn leads to the possibility of scheduling (re-ordering) of service requests (fig. 4).

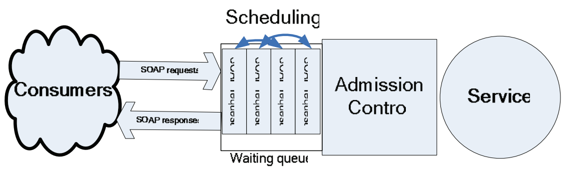

Figure 4: Transparent Scheduling.

Scheduling of requests opens a broad spectrum of possibilities, like maximizing service performance in terms of interactions per time unit, minimizing variance of service response times, etc.
This paper focuses on SJF (Shortest Job First) scheduling as a means for optimizing overall throughput. SJF (Shortest Job First) is a scheduling policy which minimizes the response time of light requests, for the price of the heavier ones. All incoming service calls are put in a waiting queue and are executed in the order of their size as shown in figure four. Smith (Smith 1956) proved that SJF is the best scheduling policy for maximizing the throughput if perfect job estimation is available.

In this paper we limit the discussion of scheduling to SOAP encoded, synchronous RPC style, stateless Web Services. In addition rather than using a complex resource model, requests (jobs) will be characterized by the load they create on the provider (server). The SJF scheduler is highly dependent on a correlation value of predicted and observed job length – the better the correlation the better the optimality of the schedule. While a correlation close to one (perfect predictions) achieves the *optimal schedule* (Conway 1967), a correlation close to minus one (always wrong predictions) results in the *worst schedule* (instead of minimizing the response time, it will be maximized). A correlation of zero (random guess, equal amounts of correct and wrong predictions) leads to a random scheduling (Conway 1967).

The SJF scheduler is highly dependent on a correlation value of predicted and observed job length – the better the correlation the better the optimality of the schedule.

3 TPC-APP

To achieve a realistic and domain independent evaluation of the transparent scheduling, the TPC-APP benchmark of the Transaction Processing Performance Council (TPC) was chosen. According to TPC (TPC 2006) the “TPC Benchmark™ App (TPC-App) is an application server and web services benchmark. The workload is performed in a managed environment that simulates the activities of a business-to-business transactional application server operating in a 24x7 environment…..” (TPC-APP 2006).

Since there were no free test-suites available, a reimplementation of the benchmarks was implemented following the TPC specifications.

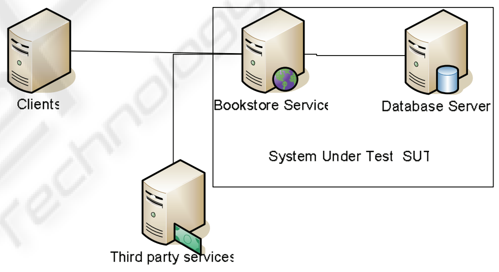

Figure 5: Topology of TPC-APP.

As shown in figure 5, the TPC-APP scenario consists of clients, a third-party service, a bookstore service and a database server. The bookstore services and the database server are hosted on different machines (application server & DB server). TPC-App also introduces a third-party service (e.g. credit card service) to simulate external parties.
The application server (bookstore service) exposes eight different methods shown in table one with their distribution in the client requests (e.g. 50 % of all client calls are Create Order requests). Two methods are Writes (Create Order, Change Item), one is Read/Write (New Customer, Change Payment) and three are Reads (Order Status, New Products, and Product Detail). TPC-APP dictates that each of these methods should be treated as a transaction with ACID properties.

Table 1: Services.

Method	Distribution
New Customer	1.00%
Change Payment Method	5.00 %
Create Order	50.00 %
Order Status	5.00 %
New Products	7.00 %
Product Detail	30.00 %
Change Item	2.00 %

The database contained all the information operated by service, like customers, orders, inventory, etc. The size of the database was 80 Mb. The inventory table was scaled to 100 000 records, and the table describing orders and clients was proportional to the number of virtual clients. In order to reduce proxy complexity component inserted between clients and application server, the underlying communication protocol was changed from the HTTPS to HTTP.

4 EXPERIMENTS

Our TPC-APP implementation uses JSE as the officially allowed by TPC platform. The application server is *Jakarta Tomcat 5.0.28* running on JSE 1.4. Tomcat's default configuration is set to use a load balancing package. Load balancing was disabled to ensure a consistent behaviour for the experiments (it also leads to a 15 ms speedup in average on each service call). Axis 1.3 served as framework for implementing the Web Services. All Web Services were implemented in a synchronous way and use the default HTTP 1.0 as the transport protocol. *MySQL 4.1.12a* was selected as the database server and the Connector J3.1.12 JDBC driver was used to link the services to the database. The application server, database server and virtual clients resided on separate machines of following configuration: Pentium IV 2.8 GHz, 2Gb of RAM. The third part services used Pentium 3, 600Mhz with 512 of RAM. The machines were connected with a 100Mb LAN and used XP SP 2 as their OS. The XP performance counters were used to collect the performance data.

As the main metrics we chose throughput and average response time. The average response time is the mean value of all the response times of service during the measurement interval. Throughput is here defined as the number of successful service interactions per time unit. The measurement interval is 30 minutes.

To emulate the behaviour of multiple clients, a workload generator is used. The settings of the client session lengths were distributed according a discrete Beta distribution with shape parameters 1.5 and 3.0. (TPC-APP 2006). In order to keep the number of simultaneous clients constant, new clients arrive after old clients have finished their session.

According to TPC-APP the client's business logic is ignored (takes zero time). Every client starts a new service call immediately after obtaining the results from the previous ones. The load on the service provider is a result of the number of simultaneous clients. HTTP 1.0 is the used transport protocol (closing the connection after each interaction). The timeout for the connection is set to 90 seconds.

The clients are simulated by the workload generator and execute a sequence of requests that is determined by invoking a random generator at runtime (the setting represent the distribution shown in table one). The process of determining calls sequences has *no memory* and consequently the probability of the next operation being invoked does not depend on the previous call.

Two types of experiments were conducted. The first experiment is used determine if posteriori knowledge (runtime statistics) can predict with sufficient *accuracy* the behaviour of a job. The second type of experiments was used to determine the *impact* of scheduling.

5 EVALUATION

5.1 Job-Size Estimation

In order to evaluate the sizes of the jobs we divided all the SOAP requests into classes. SOAP requests within the same class create approximately same loads on the service. Using the average response time of the class it is possible to predict the behaviour of a request.

Trace studies of the TPC-APP benchmark workload show that a classification according to the name of a service operation allows achieving only a (weak) 0.35 correlation between the estimated and the actual response times. As can be seen in figure six, some operations ("Order Status", "New Products") have a high variation of execution time, while operations, like "Change Payment", exhibit more stable behaviour. However, if the data contained in the SOAP messages is also used for classification a (strong) 0.72 correlation between the estimated and the actual response times is achieved [fig. 7]. The reason for the strong correlation is an effect of using a database centric scenario in which the costs of a service call relate to the number of updates/writes performed on the database.

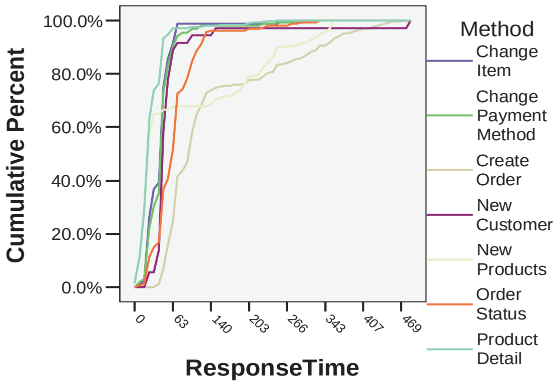

Figure 6: Execution time of service methods (16 clients).

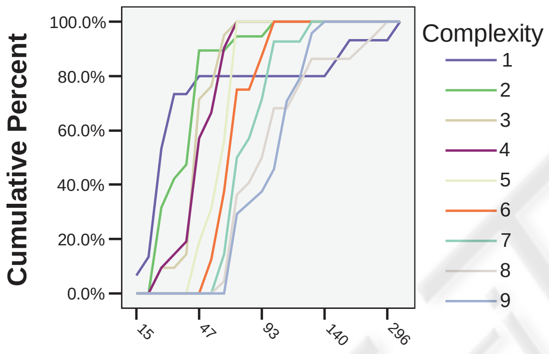

Figure 7: Execution time of "Order status" with different SOAP request complexity (16 clients).

5.2 Admission Control & Scheduling

To obtain the data of a SOAP message is necessary to perform a partial request parsing. In order to minimize scheduling overhead and avoid double parsing we located the proxy component into the Tomcat framework (proxy resides on the same machine as app-server).

Figures 9 and 10 show the observed throughput and response time in relation to used policy {PS, FIFO, SJF} and number of simultaneous clients.

The first observation is that even with admission control (FIFO) and SJF, a decline in throughput can be observed. The decline is a result of the thrashing due to overload of their parsing parts. The overload situation the admission control and scheduler experience and can be easily solved/eased by hosting these components on a more resource rich host. In the current setting, the proxy that performs the scheduling/admission control is residing on a host that is shared with the application server [fig. 8.].

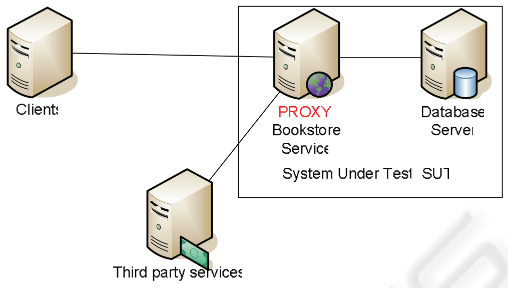

Figure 8: Topology of the scheduled environment.

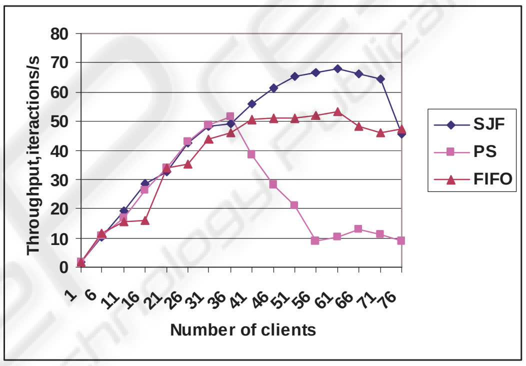

Figure 9: Throughput of the service governed by various scheduling policies.

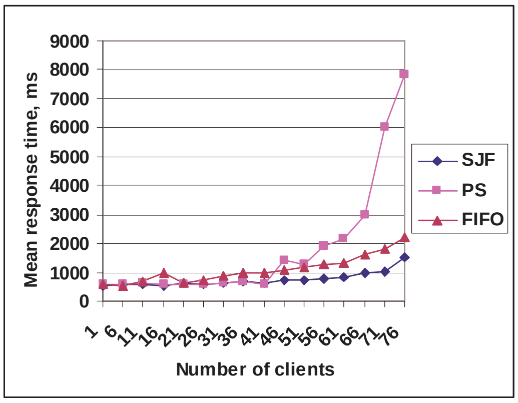

Figure 10: Response times of the service governed by various scheduling policies.

The second observation is the significant performance boost FIFO and SJF achieve in overload situations. While PS degrades as expected (at ca. 41 clients) SJF and FIFO begin to degrade only at around 76 clients.

Until the load reached its peak values (41 users) the buffer queue created by the admission control is mostly empty therefore all scheduling policies behave in the same manner.

Table 2: Application server loads with 36 clients.

Policy	Original	FCFS	SJF
CPU, %	63	40	37
RAM, MB	557	557	558
Network Bytes/sec	271220	261629	73750

Table 3: Database server loads with 36 clients.

Scheduling policy	Original	FCFS	SJF
CPU, %	3.44	13.8	14.21
Memory, Mb	1.45	1.4	1.39
Network, Kbytes/sec	55.97	19.39	20.90
Reading, Mb/sec	0.11	0.32	0.34
Writing,Mb /sec	0.65	0.13	0.13

After the load exceeded its peak value the excess requests were buffered and the throughput was preserved. Further load growth created a situation in which there were more elements in the queue to be scheduled so SJF and FIFO began to behave different. At higher loads SJF started outperforming FIFO by 20% in average. Table two shows that scheduling did not affect the consumption of the resources like, memory or network. Meanwhile the CPU utilization went down from 63% to 37-40% due a lesser amount of parallel jobs. In general, low values of CPU utilizations are the result of using HTTP 1.0 since establishing a new socket causes a significant delay (up to 1 sec) and does not require the processor resources. It can also be seen that the database server's CPU load and amount of reads/writes had highest values in case of SJF [table 3]. Thus even these loads were not fully utilizing the database. The under-utilization of the database can be explained by the low frequency of database queries issued by application server. Consequently, the application server is the bottleneck. In the current setup scheduling of the service requests mostly affected the performance of the bottleneck element – the application server.

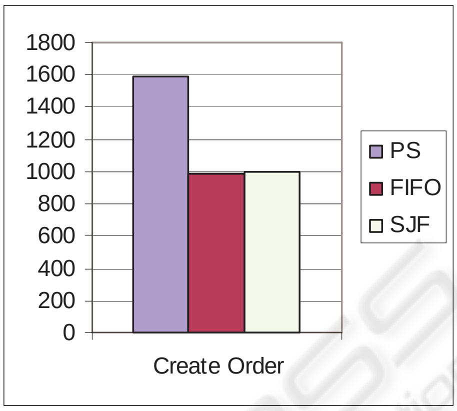

Figure 11: “Create Order” Operation.

As a result of scheduling, the application server was able to produce database queries faster. Thus the database server utilization (CPU, read/write operations) increased by 10% [tabl. 3]. Applying scheduling caused an interesting effect of “resource load equalisation” in which the usage of overloaded resources decreased, while the utilization of lesser used resources increased. Nevertheless, this is an observed effect that requires research.

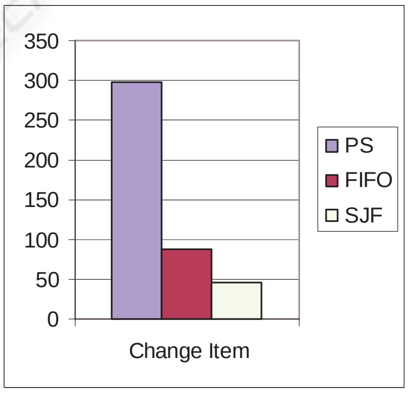

Figure. 12.”Change Item” Operation.

It is important to note, that the skew of the response time distribution was small [fig. 13] and that the difference between the slowest and the fastest request reached only the ratio of one to six. Therefore the possibility of optimizations was lower and the bigger jobs were penalized only lightly.

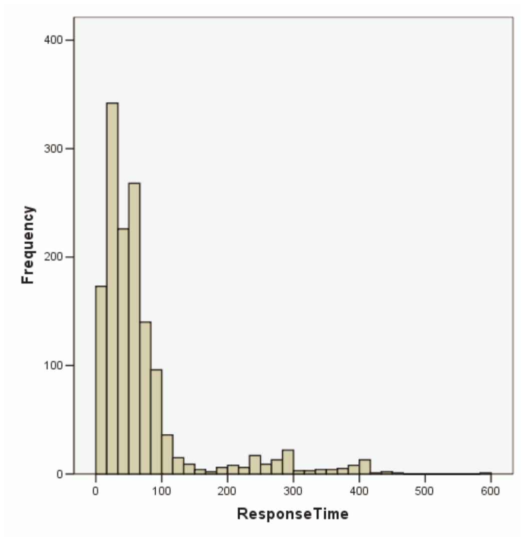

Figure 13: The distribution of the service response times (without network overhead).

Figures 12 and 13 describe an example of a light and a heavy request in a medium load case (11 clients). SJF to FCFS heavy jobs experienced the 2% increase of the response time, while the response time of smaller jobs shrunk up to 45%. In any case, each of these policies significantly outperformed default PS.

6 RELATED WORK

Elnikety et. al. (Elnikety 2004), present a method for admission control and request scheduling for multi-tired e-commerce Web-sites with a dynamic content in. The authors use a transparent proxy, as the intermediate scheduling component between application server and database server. The method enabled overload protection and preferring scheduling SJF augmented with aging method.

SJF scheduling has been successfully used for http requests by Cherkassova in (Cherkassova 1998). This work proposes the use of Alpha-scheduling to improving the overall performance. Alpha-scheduling, is a modified SJF algorithm and prevents the starvation of big requests, in way similar to aging methods (Elnikety 2004). The job size in (Elnikety 2004) is estimated by classifying all http requests according a name of the requested servlet.

In the domain of web services admission control research has focussed on supporting QoS differentiation (providing different levels of QoS). Siddaharta et al. (Siddaharta 2003) introduce the concept of using a Smartware platform as managing infrastructure for Web Services. The Smartware platform consists of three core components namely interceptor, scheduler and dispatcher. The interceptor intercepts the incoming requests, parses them and classifies them according to user and client (device) type. Once classified the requests are subjected to a scheduling policy the scheduler determines the order of execution. Finally the dispatcher routes the request to the service endpoint. Smartware implements a static scheduling by assigning priorities to service calls according the classification of the interceptor. As a main scheduling means the Smartware authors chose a randomized probabilistic scheduling policy namely lottery scheduling (Waldspurger 1994).

Sharma et al (Sharma 2003) achieve QoS differentiation using predefined business policies. The authors examined static scheduling, in which the request class is determined by the application type, device type and client (user) level. The overall priority is calculated on the base of component priorities, for example it can be a sum or a product. The problem of starvation was eliminated by applying probabilistic scheduling. However, static scheduling does not appear to be a capable for ensuring differentiated throughput in case of fluctuating loads (Sharma 2003). The authors suggest augmenting the static scheduling by altering priorities at runtime using the observed throughput. The adjustments to the required throughput are done using a correcting factor that is calculated as a penalization function of a hyperbolic nature, which speeds up slow responses and slows down fast responses in compliance with each request class.

7 CONCLUSION

This paper presents the idea of transparent scheduling of Web Services requests as a means for achieving better performance. Using the TPC-APP, a two-tier B2B application, as a benchmark we evaluated the performance gains of SJF (Shortest Job First) compared to FIFO admission control and standard PS. By simply adding a proxy between consumer and provider, it was possible to achieve a transparent scheduling and admission control that lead to significant performance improvements in overload cases. In addition, the experimental evaluation showed that even in the absence of a priori knowledge, a SJF scheduler that uses observed runtime behaviour can lead to schedules that outperform FIFO and PS, making it an attractive approach for boosting Web Services performance. The results of the experimentation indicate that transparent scheduling can be applied to Web

Services as an effective and easy to implement approach for boosting performance and avoiding service provider trashing. And while SJF does penalize larger jobs, this doesn't seem to exceed 10 % which seems acceptable given the overall gains.

8 FUTURE WORK

While the results of applying scheduling are very promising it is important to note that the current work only focused on a very simplified SOA architecture. Future work in transparent scheduling of Web Services will overcome this by addressing the following issues.

8.1 Document-Style

In the current work we focused on RPC-style Web Services, that exhibit the basic request/response MEP (Message Exchange Pattern). Document-style interaction supports more complex MEPs and raises new question in regards to scheduling.

8.2 Composite Services

Composite Services orchestrate the functionality provided by the other services thus creating complex environment with significantly more complex behaviour. We hope that by applying scheduling to this environment it should be possible to increase services dependability, performance, etc.

8.3 Service-Level Agreements (SLA)

SLAs are an increasingly important aspect of SOA. Scheduling can be used as a means for achieving this by minimizing penalties and supporting QoS contracts in critical situations.

ACKNOWLEDGEMENTS

This research has been supported by NSERC grants and the equipment provider by the Canadian Foundation for Innovation (CFI).

REFERENCES

Apache, 2006. Available at: http://httpd.apache.org/

Axis, 2006, Available at: http://ws.apache.org/axis/

Box, D., 2004, "Four Tenets of Service Orientation", Available at: http://msdn.microsoft.com/msdnmag/issues/04/01/Indigo/default.aspx

Christensen, E., Curbera, F., Meredith, G., Weerwarana S.,, 2006, Available at: http://www.w3.org/TR/wsdl.

Cherkasova, L., "Scheduling strategy to improve response time for web applications",1998, in HPCN Europe ,Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking. London, UK: Springer-Verlag, pp. 305–314.

Conway, R., W., et al. 1967.. Theory of scheduling, Addison-Wesley, Massachusetts, USA, 1967.

Elnikety, S., Nahum, E., Tracey, J., Zwaenpoel, W., , 2004. "A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites". In Proceedings of the 13th international Conference on World Wide Web (New York, NY, USA, May 17 - 20, 2004). WWW '04. ACM Press, New York, NY, pp. 276-286.

Extensible Markup Language (XML), Available at: http://www.w3.org/XML/

Eclipse, 2006. http://www.eclipse.org/

Graham, S., Davis, D., Simoenov, S., Daniels, G., Brittenham, P., Nakmura, Y., Fremantle, P., Konig, D., and Zentner. 2004.C. Building Web Services with Java. Sams Publishing, Indianapolis, Indiana, USA.

Heiss, H.-U. , Wagner, R., 1991. "Adaptive load control in transaction processing systems", In 17th International Conference on Very Large Databases, Barcelona, Spain.

Mitra, N., 2006. SOAP version 1.2 part 0. Available at: http://www.w3c.org/TR/soap12-part0/.

Sharma, A.,, Adarkar, H., Sengupta, S., 2003 "Managing QoS through prioritization in web services," WISEW, vol. 00, pp. 140–148.

Siddhartha, P., Ganesan, R., Sengupta, S., 2003, "Smartware - a management infrastructure for web services." in WSMAI, pp.. 42–49.

Smith,W.,E., 1956. Various optimizers for single-state production. Naval Research Logistics Quarterly, 1956.

TPC, 2006. Transaction Processing Performance Council, Available at: http://www.tpc.org/.

TPC-APP, 2006. Available at: http://www.tpc.org/tpc_app/

Visual Studio, 2005. Available at: http://msdn.microsoft.com/vstudio/

Waldspurger, A., Weihl, W., E., 1994, "Lottery scheduling: Flexible proportional-share resource management," in Operating Systems Design and Implementation, pp. 1–11.

W3C, 2006, Available at: http://www.w3.org/.

XML, 2006. Available at; http://www.w3.org/XML/.

XML-RPC, 2006, Available at: http//www.xmlrpc.com/spec.