Serdica J. Computing **1** (2007), 267–278

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

TRANSPARENT SCHEDULING OF COMPOSITE WEB SERVICES*

Dmytro Dyachuk, Ralph Deters

Abstract. Composite Web Services (CWS) aggregate multiple Web Services in one logical unit to accomplish a complex task (e.g. business process). This aggregation is achieved by defining a workflow that orchestrates the underlying Web Services in a manner consistent with the desired functionality. Since CWS can aggregate atomic and other CWS they foster the development of service layers and reuse of already existing functionality. An important issue in the deployment of services is their run-time performance under various loads. Due to the complex interactions of the underlying services, a CWS they can exhibit problematic and often difficult to predict behaviours in overload situations. This paper focuses on the use of request scheduling for improving CWS performance in overload situations. Different scheduling policies are investigated in regards to their effectiveness in helping with bulk arrivals.

1. Introduction. Well defined, loosely coupled services are the basic building blocks of the service-oriented (SO) design/integration paradigm [1].

ACM Computing Classification System (1998): H.3.5; F.2.2.
Key words: Web Services, Composite Web Service, LWKR, SJF, Scheduling, Admission Control.

*The paper has been presented at the International Conference Pioneers of Bulgarian Mathematics, Dedicated to Nikola Obreshkoff and Lubomir Tschakaloff, Sofia, July, 2006.

Services are computational elements that expose functionality in a platform independent manner and can be described, published, discovered, orchestrated and consumed across language, platform and organizational borders. While service-orientation (SO) can be achieved using different technologies, Web Services (WS) is the most commonly used one due to the standardization efforts and the available tools/infrastructure.

Using WS it is fairly easy to expose existing applications/resources and combine them into novel services. However, the ease with which legacy systems can be aggregated/orchestrated, leads to the questions of how the new Composite Web Service (CWS) perform in overload situations.

This paper focuses on the use of request scheduling and admission control as mechanisms for improving CWS performance in overload situations. Two scheduling policies namely Shortest Job First (SJF) and Least Work Remaining (LWKR) are investigated in regards to their effectiveness in helping with bulk arrivals.

2. Overload behaviors. Web Services are most often used to expose some already existing legacy functionality (e.g. transactional database). If service providers do not share resources (e.g. no two providers expose the same database), than every service provider can be modeled as a separate service. If these providers are capable of handling multiple requests simultaneously, then each incoming request must be handled by a separate thread. Since each service has a finite amount of resources it can only serve a certain number of requests simultaneously (actual number depends on the job sizes of the requests). Fig. 1 shows the (simplified) behavior of services under various loads.

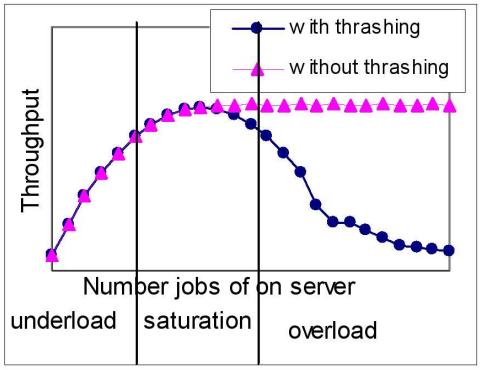

Fig. 1. Service behavior under various loads

If a service is gradually exposed to an ever-increasing number of service requests three distinct stages, namely *underload, saturation* and *overload* can be observed. In the beginning the service experiences a load that is below its capacity (*underload*) and consequently it is not fully utilized and when the number of requests is increased the throughput (number of completed jobs per time unit) increases. As the rate of incoming requests continues to increase, the server will reach its *saturation* point (peak load). This 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 request arrival rate leads to an overload and ultimately the *thrashing effect* [2]. Thrashing occurs either as a result of an overload of physical resources (*resource contention*) like processor or memory or as a result of locking (*data contention*).

Thrashing is particularly problematic in workflows since the throughput of the flow in the network is equal to the throughput of the "slowest" intersection. Consequently trashing of one service will therefore negatively impact the performance of the CWS.

3. Adaptive load control and scheduling. Since transient overload situations lead to a decline in service throughput, it is important to avoid them. Heiss and Wagner [2] proposed the use of *adaptive load control* as a means for preventing overloads, by first determining the maximum number of parallel requests (e.g. maximum number of simultaneous consumers) and then buffering/ queuing new requests once the saturation point has been reached. The impact of this approach can be seen in Fig. 1. The darker curve (circles) shows the characteristic three phases an uncontrolled server can experience, underload, saturation and overload. Using an admission control (grey, triangles), the thrashing is avoided due to the limiting the number of concurrent threads and buffering/queuing of requests above peak load.

3.1. Admission control for web services. Adding admission control to an already existing service (e.g. WS, CWS) can be achieved by use of the proxy pattern [3]–[7]. As shown in Fig. 2, a proxy that shields/hides the original provider enables the introduction of a *transparent* admission control.

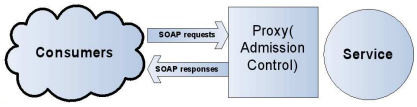

Fig. 2. Transparent admission control

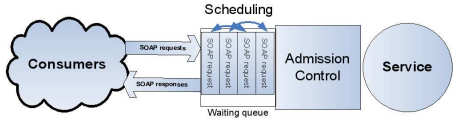

Fig. 3. Transparent scheduling

The role of the proxy is to monitor the rate at which consumers issue requests and to prevent overloading. When the request rate exceeds the capacity of the provider, a FIFO queue is used to buffer the excess requests. Transparent admission control is an effective approach for handling requests bursts [5, 8], especially if the requests impose similar loads on the service. However, as soon as the job size (impact on service) of requests varies, a FIFO queue is no longer sufficient and reordering (scheduling) is required to ensure that the service is neither overloaded nor underutilized.

3.2. Scheduling of requests. Since SO middleware (e.g. Web Services) tend to foster declarative communication styles (e.g. SOAP[9]), it is fairly easy to analyze the service bound traffic, identify the request and estimate the impact each request will have on the service provider (job size). This in turn enables re-ordering (scheduling) of the service requests. Scheduling of requests (Fig. 3) opens a broad spectrum of possibilities, like maximizing service performance in terms of interactions per time unit and/or minimizing the variance of service response times, etc.

For an atomic (non composite) service SJF (Shortest Job First) scheduling is sufficient as a means for optimizing overall *throughput*. SJF is a scheduling policy which minimizes the response time of light requests, at the expense 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 Fig. 3. Smith [10] proved that SJF is the best scheduling policy for maximizing the throughput if accurate information on job sizes is available.

3.3. Scheduling & composite web services (CWS). Scheduling of CWS requests can be done on a *system* (workflow/CWS) or *component* (service) level. While system-level scheduling treats the CWS as an atomic entity and thus reorders the requests *prior* to invoking the workflow, component-level scheduling focuses on scheduling the sub-requests for each service and can adjust to changes during the execution of the workflow. In this paper only Composite Web Services composed according to the *Sequence* pattern [11] with a static structure and known job-size are considered.

Since system-level scheduling treats the CWS as one component, SJF is suitable. SJF reorders the incoming requests according to their execution time, by placing the shortest requests in the beginning of the queue, while the largest are put at the end. The execution time for a CWS request is the sum of the times needed for processing the sub-requests on each service.

In component-level scheduling a separate scheduler is placed in front of each service resulting in a multi-step scheduling. In case the component

schedulers can only estimate the size of the sub-request assigned to them, SJF should be used. However, if each component scheduler can evaluate the size of the remaining work in the workflow or the mentioned above information can be shared among all the schedulers, LWKR (Least Work Remaining) can be used. LWKR evaluates the total cost of all remaining sub-request and assigns the priorities according to the remaining work. Since LWKR scheduling is performed in every component scheduler, the costs of scheduling are higher but at the same time the scheduling can be adapted better to unexpected outcomes of processing requests.

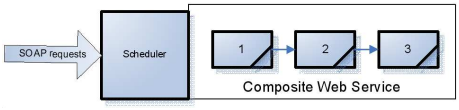

Fig. 4. System-level scheduling

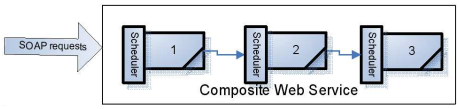

Fig. 5. Component-level scheduling

4. Experiments. To test if thrashing can also be observed when lightweight services are deployed a simple WS named "echo" was developed (using Axis [12]) and tested under various loads. The service "echo" contains a single method that expects an array of 200 integers (ranging between 0 and 5) as input and returns an array of 200 integers consisting of the corresponding Fibonacci numbers. Each experiment run lasts about one hour. The clients are sending requests at a rate varying from ten requests per second to eighty five requests per second. The collected data includes the number of page faults per second, CPU utilization, number of threads on server, amount of memory used by application server and the number of requests being handled concurrently.

The throughput chart in Fig. 6.a shows that the "echo" service can also experience thrashing. The throughput of the service grows proportionally with the request rate, until the service is overloaded. After the request rate exceeds sixty five, the growth slows down since the service entered its saturation phase. Further load increases cause a steep drop in the throughput of the system since it becomes overloaded. The same can be observed in the response time diagram in Fig. 6.d. While the service is in an underload phase the execution time of the requests is relatively constant (variation is in the range of 34–38 milliseconds). However, it starts increasing as soon as the system becomes saturated. The values rise from 93 ms to 1 s, when the load reaches 60–70 requests per second. In the overload phase the service response time jumps up to 11–30 seconds (a 1000 times increase compared to underload). Each test is executed for a limited period of time. The experiments with overload cases are lasting one minute each, indicating the heavy influence of even short-lived heavy loads (bursts). The charts

in Figs 6.b and 6.c depict the load on the main resources utilized by this service indicating that parsing introduces a heavy load on CPU and memory.

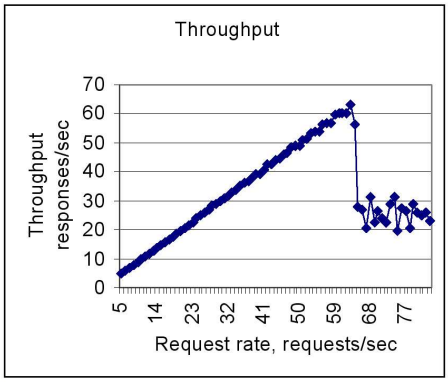

Fig. 6.a. Throughput

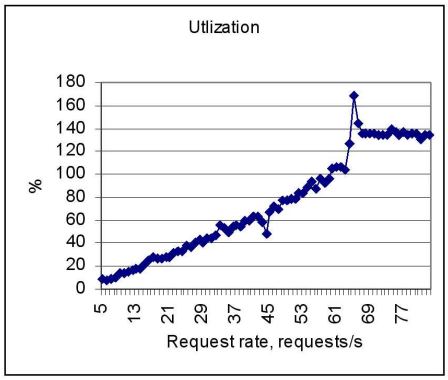

Fig. 6.b. CPU Utilization

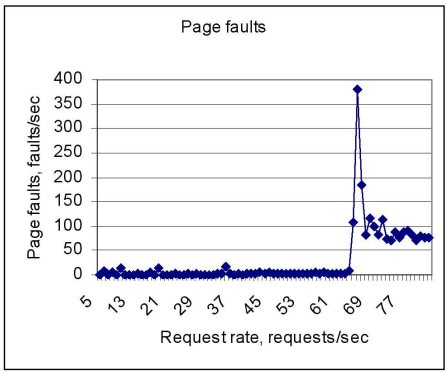

Fig. 6.c. Page faults

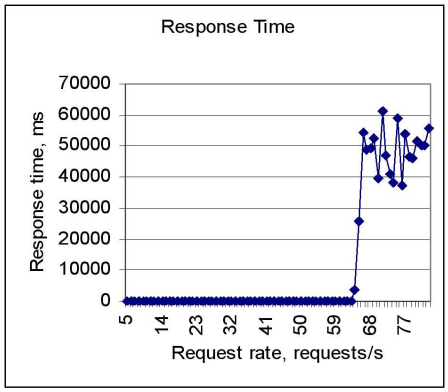

Fig. 6.d. Response time

As can be seen the in the underload and saturation phases, utilization of the CPU is proportional to the load created by the clients while memory usage stays constant. As the load reaches its peak value, the memory consumption starts to increase drastically, which leads to a high page fault rate. Meanwhile the CPU consumption drops, since threads are waiting for memory operations to be competed. This causes delays in executing the jobs and as the new requests are arriving an accumulation of the jobs in the service begins that leads to a vicious circle of decreasing job execution and increased job accumulation.

4.1. Simulation of web service. Studying the impacts of overloads (e.g. bursts) on CWS leads to the problem of running multiple services in a

controlled environment which is very resource intensive. Simulation of services offers a less resource intensive alternative for studying the behavior of CWS and was consequently chosen in this research. Using the simulation tool AnyLogic [13], a model that captures the behaviors of WS and CWS was developed [5]. The previous experiments showed that thrashing appears not only in WS running a transactional database at the backend, but also in the standalone computation bound WS as well. Thrashing in the "echo" service originates from the parsing, which is mandatory part for each WS. Therefore all the WS without admission control will experience the thrashing effect if the loads exceed certain limits.

To calibrate the simulation the WS "echo" from the previously described experiment was used. The WS model contains two main resources CPU and memory, governed by PS and tries to accurately describe the observed behavior (e.g. page faults). The model has been designed to exhibit the same trashing effects that have been observed in the experiments. Figs 7.a and 7.b present a comparison of the simulated and observed Axis WS behavior indicating a close match.

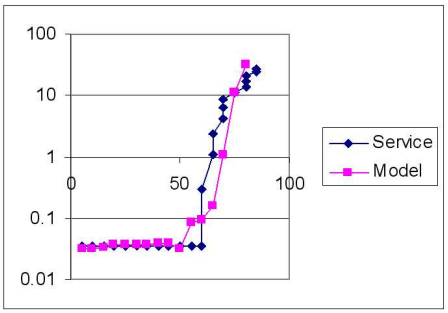

Fig. 7.a. Model and echo service response time

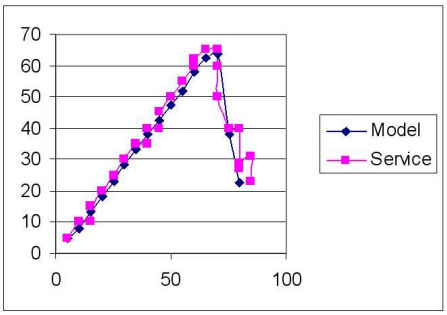

Fig. 7.b. Model and echo service throughput

4.2. Scheduling of CWS. After ensuring that the simulated "echo" service demonstrates the same behavior as the Axis WS, experiments with simulated CWS were possible. Three WS of identical capacity were aggregated (sequence workflow pattern) into one CWS. Invocation of the CWS leads to the sequential invocation of Service 1, Service 2 and finally Service 3. To simplify the simulation the costs for orchestration are assumed to be zero. Therefore the time between receiving a response from a prior service and sending a request to a next service is neglected. The upper limit on the number of the concurrent jobs per service is set to 10.

The job-size of the client request for the CWS exponentially distributed

with an average at 340 ms. A high service request rate (50 requests/sec) is alternated with a low request rate (2 requests/sec) (Fig. 8). The duration of the low load is 480 seconds and 20 seconds for a high load.

4.3. Impact of Scheduling on CWS. The services without admission control are "open" to the burst arrivals. As a result of the high load period the requests get accumulated at Service 1. Consequently the high level of the concurrently executed threads causes the thrashing effect, which results in a low throughput with average value 4-6 responses per second (Fig. 8.a). Please note that the throughput of the first service is the arrival rate of the second service. Since the second service has an identical capacity, it is not overloaded and consequently its throughput mirrors the arrival rate. The same situation is observed with the third service. Hence CWS and Service 1 throughputs had the same value (Fig. 8.e). The accumulation of the jobs in Service1, caused by the absence of the admission control mechanisms results in the increase of its service time. The service time grows with the number of jobs and as soon as the arrived jobs start departing the response time decreases (Fig 8.c).

With LWKR due to the limitation on the number of concurrent jobs the service response time has a more uniform nature (Fig. 8.d) and peaks disappear faster.

Placing a LWKR scheduling proxy in front of Service 1 decreases the number of threads (handling requests concurrently) to 10. As a result the throughput goes up to 10–12 responses per second (Fig. 8.b). Since Service 2 and Service 3 have the same capacity as Service 1, their and CWS's throughput are very similar (Fig. 8.f).

As Service 1 is the first service in the chain, it is exposed to the highest load and its response time has the highest value. As a result it has the biggest impact on the response time of the workflow. The charts in Figs 8.g and 8.h show that the shape of the response time curves is repeating the shape of the curves from charts in Figs 8.c and 8.d.

The bar diagram on the Fig. 9.a. depicts the average response time of the CWS. Applying admission control decreases the average service time from 145.22 to 74.87 seconds. Scheduling of CWS as a single unit (SJF) reduces the response time by 36% when compared to the FIFO scheduling policy. In SJF the decisions regarding the jobs order are made at the moment of their admission into the CWS, thus the schedule is made in correspondence to the sizes of the jobs at each service. Since the requests go through Service 1, their order remains almost unchanged, thus for Services 2 and 3 it can be non-optimal. This issue does not appear in LWKR which considers the amount of the reaming work for

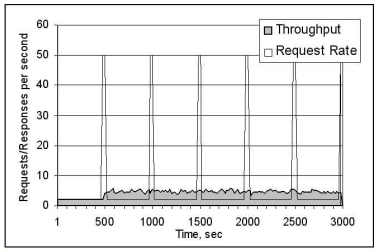

Fig. 8.a. Service 1 Throughput. No admission control

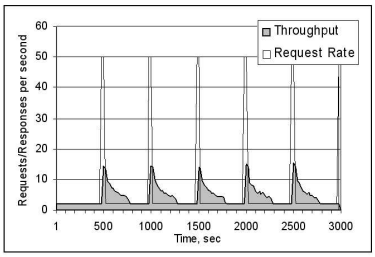

Fig. 8.b. Service 1 Throughout. LWKR

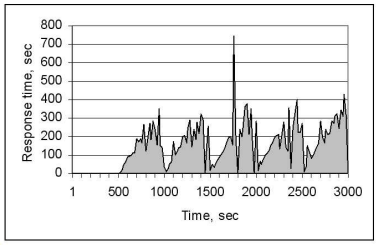

Fig. 8.c. Service 1 response time. No admission control

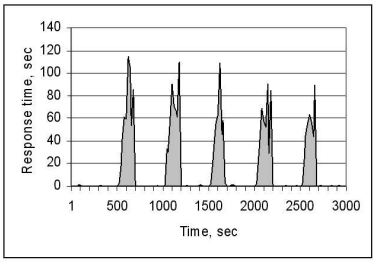

Fig. 8.d. Service 1 response time. LKWR

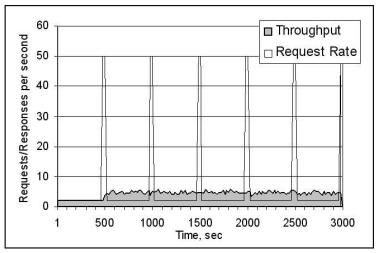

Fig. 8.e. CWS throughput. No admission control

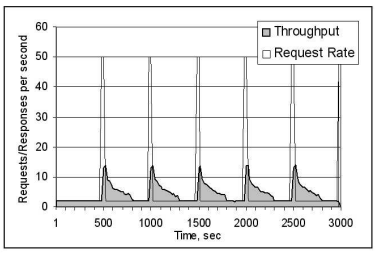

Fig. 8.f. CWS throughput. LWKR

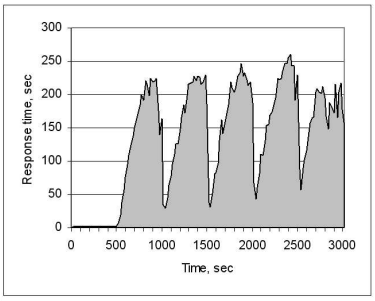

Fig. 8.g. CWS response time. No admission control

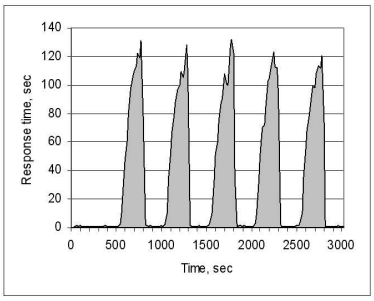

Fig. 8.h. CWS response time. LWKR

each request before their execution. As the result LKWR outperforms SJF by 7%. In SJF, the priorities of the CWS requests are determined by calculating the sum of all components services times. For LWKR the first sequence of processing requests is determined in the identical way but the requests are also scheduled at Services 2 and 3. Nevertheless, SJF exhibits almost the same performance increase as the LWKR, with a lesser amount of control over the services. The reason is that the throughput of the Service 1 is not sufficient for creating a big buffer queue at Services 2 or 3. In consequence scheduling possibilities are reduced and all the incoming requests for Services 2 and 3 most likely are processed in the same way as Service 1.

It is noteworthy that the component SJF (CSJF) improvements are outperformed by the system SJF. System SJF sacrifices the performance of the first service for the sake of improving overall performance. Component SJF has a 32% better performance for Service 1 (Fig. 9.a) and a worse one for the next services.

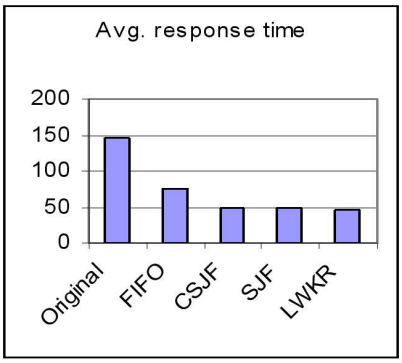

Fig. 9.a. CWS average response time

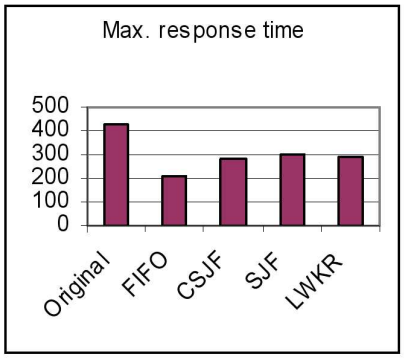

Fig. 9.b. CWS maximum response time

It is interesting to note that in terms of maximum response time FIFO outperforms all the other scheduling policies (Fig. 8.b). LWKR, CSJF, SSJF are giving priority to "smaller" jobs, thus penalizing the larger jobs. Light penalization of the small number of the bigger jobs, leads to a decrease in the service time for the smaller jobs. Nevertheless the penalization of large jobs does not exceed the service time values of the original configuration without admission control (Fig. 9.b).

5. Conclusions & future work. This paper presents the idea of transparent scheduling of Composite Web Services requests as means for achieving better performance. In order to evaluate CWS scheduling policies a model for simulation was developed. The simulation shows that scheduling significantly

reduces the average response time for the CWS in case of bulk arrivals. The experiments showed the benefits of applying scheduling to CWS that orchestrate according to the sequence workflow pattern.
LWKR improves the performance up to 68%, but requires placing a proxy in front of each component. Using SJF and having just one scheduler for the workflow leads to a 7% drop in performance compared to LWKR. 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:

- **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.
- **Composite Web Services with more complex workflow patterns:** Scheduling Composite Web Services that implement different workflow patterns is still an open issue.
- **Service-Layer Agreement** [14] **(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.

REFERENCES

[1] Four Tenets Of Service Orientation. `http://msdn.microsoft.com/msdnmag/issues/04/01/Indigo/ default.aspx`.

[2] Heiss H., R. Wagner. Adaptive load control in transaction processing systems. In: Proceedings of the 17th International Conference on very Large Data Bases, VLDB'91, (1991), 47–54.

[3] Dyachuk D., R. Deters. Optimizing performance of web service providers, In: Proceedings of IEEE International Conference on Advanced Information Networking and Applications, AINA-2007, 2007.

[4] Dyachuk D., R. Deters. Scheduling of composite web services, In: Proceedings of OTM 2006 Workshops on The Move to Meaningful Internet

Systems, Heidelberg, Lecture Notes in Computer Science, Vol. **4277**, Springer-Verlag, 2006, 19–20.

[5] Dyachuk D., R. Deters. Transparent scheduling of web services. In: 3rd International Conference on Web Information Systems and Technologies, 2007.

[6] Erradi A., P. Maheshwari. wsBus: QoS-aware middleware for reliable web services interactions. In: EEE'05: Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05) on e-Technology, e-Commerce and e-Service, 2005, 634–639.

[7] Siddhartha P., R. Ganesan, S. Sengupta. Smartware – A management infrastructure for web services. In: WSMAI, 2003, 42–49.

[8] Elnikety S., E. Nahum, J. Tracey, W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In: WWW'04: Proceedings of the 13th International Conference on World Wide Web, 2004, 276–286.

[9] Mitra N. SOAP version 1.2 part 0, 2007. `http://www.w3c.org/TR/soap12-part0/`.

[10] Smith W. E. Various Optimizers for Single-State Production. Naval Research Logistics Quarterly, 1956.

[11] Aalst W. M. P. V. D., A. H. M. T. Hofstede, B. K. a. A. P. Barros. Workflow Patterns, *Distrib. Parallel Databases*, **14** (2003), 5–51.

[12] Apache Axis. `http://ws.apache.org/axis/`.

[13] XJ Techologies. Anylogic 5.5. `http://www.xjtek.com/`.

[14] Keller A., H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. IBM Research Report, May, 2002.

Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan
S7N 5C9 CANADA
e-mail: `dod401@mail.usask.ca`
e-mail: `deters@cs.usask.ca`