PAPER 3187: PETSTORE-WS: MEASURING THE PERFORMANCE IMPLICATIONS OF WEB SERVICES*

Kai S. Juse, S. Kounev and A. Buchmann

Department of Computer Science
Darmstadt University of Technology, Germany
{ksjuse,skounev,buchmann}@informatik.tu-darmstadt.de

Web Services are increasingly used to enable loosely coupled integration among heterogeneous systems but are perceived as a source of severe performance degradation. This paper looks at the impact on system performance when introducing Web Service interfaces to an originally tightly coupled application. Using two implementation variants of Sun's Java Pet Store application, one based strictly on the J2EE platform and the other implementing some interfaces as Web Services, performance is compared in terms of the achieved overall throughput, response times and latency.

1 Introduction

Web Services are a novel kind of Web applications. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Web services perform functions which can be anything from simple requests to complicated business processes [Tid00]. More precisely, a Web Service is a software entity that is made available over the Internet and uses a standardized XML messaging system for communication [Cer02]. The definition and description of the public interfaces is also done using XML [BCF+03].

Web Services facilitate application to application communication for building integrated systems of Web application components. As a member of the family of service-oriented architectures (SOA), it encourages loose coupling of participating applications. The large-scale use of XML technologies is the foundation for the platform independence of Web Services, which makes it particularly suitable for the integration of heterogeneous systems. However, the use of complex XML mechanisms is also often perceived as a potential source of Web Service performance degradation. To study the performance implications of the use of Web Services for application integration, we analyze and compare the behavior of a system using communications purely based on the Java Message Service (JMS) API to a variation of that system employing Web Services on some of its interfaces.

*This work was partially funded by BEA Systems, Inc. as part of the project "Capacity Planning and Performance Analysis of J2EE Applications and Web Services" and the Deutsche Forschungsgemeinschaft (DFG) as part of the PhD program "Enabling Technologies for E-Commerce" at Darmstadt University of Technology.

In section 2, we introduce the Java Pet Store application as the subject of our experiments. We describe its structure and important aspects of its mode of operation. Section 3 specifies the setup and environment of the experiments together with the load scenario and the performance metrics used. The measurement results are presented in section 4, followed by summary and conclusions in section 5.

2 The Java Pet Store Application

The Java Pet Store Sample Application [Sund] was developed within the Java BluePrints Program [Suna] at Sun Microsystems, Inc. It demonstrates how to use the capabilities of the J2EE platform to develop robust, scalable and portable e-business applications. It comes with full source code and documentation, allowing application developers to experiment with J2EE technologies and learn how to use them effectively to build their own enterprise solutions. Pet Store can also be considered as a reference implementation of numerous established design patterns [Sunb] and best practices for enterprise Java programming. In addition to standard J2EE interfaces, starting with version 1.3.1, Pet Store also includes Web Service interfaces to some of its services. The goal is to showcase the use of Web Services within the J2EE platform. Following is a brief description of Pet Store based on [Sund] and [SSJt02].

2.1 Business Problem Modelled

The Java Pet Store application models a typical e-business — an online store that sells products (in this case animals) to customers. Conceptually, the

business divides into two functional units: *Web Site* and *Fulfillment Center*.

The Web site presents an online interface to the store, through which customers can shop and place orders. When a customer completes an order, the latter is sent to the order fulfillment center for processing. Hence the Web site functional unit can be thought of as the *front end* of the enterprise. The fulfillment center, on the other hand, has an order fulfillment component and a supplier component. The fulfillment center processes orders based on the enterprise's business rules, manages financial transactions, and arranges for products to ship to customers. Because not all products are in stock at any given moment, order processing may occur over a period of time. Administrators and other suppliers may interact with the fulfillment center. This portion of the business is referred to as the *back end*. Although the supplier component is part of the sample application, it could just as easily be a service external to the application.

Versions of Pet Store prior to 1.3 were examples of a monolithic application that handled both customer interactions at the Web site as well as order tracking and administration. Real-world enterprise applications, however, are seldom single, monolithic systems. Most enterprise applications must cooperate with multiple external systems such as legacy databases, enterprise resource planning (ERP) systems or Web Services of business partners. Therefore, in order to make things more realistic, starting with version 1.3, the Pet Store application was refactored into separate modules and the ability to interact with multiple external suppliers was added. The result is a decoupled enterprise architecture that can interoperate with existing data sources and business partners' systems, all built on top of the J2EE platform. The new sample application comprises four separate sub-applications, each of which is a J2EE application [SSJt02]:

1. *Pet Store E-Commerce Web Site*: A Web application that shoppers use to purchase merchandize through a Web browser.

2. *Pet Store Administration Application*: A Web application that enterprise administrators use to view sales statistics and manually accept or reject orders. While being a Web application, it features a rich client that uses XML messaging in addition to a plain HTML interface.

3. *Order Processing Center (shortened "OPC")*: A process-oriented application that manages order fulfillment by providing the following services to other enterprise participants:

   - Receives and processes orders placed through the Pet Store Web Site. Orders are received as XML documents.
   - Provides the PetstoreAdmin application with order data using XML messaging over HTTP.
   - Sends email to customers acknowledging orders using JavaMail.
   - Sends purchase orders (described as XML documents) to suppliers via JMS.
   - Maintains purchase order database.

4. *Supplier Application (shortened "Supplier")*: A process-oriented application that manages shipping products to customers by providing the following services:

   - Receives purchase orders (in the form of XML documents) from the OPC via JMS.
   - Ships products to customers.
   - Provides manual inventory management through a Web-based interface.
   - Maintains inventory database.

Figure 1 depicts the most important Pet Store sub-applications and shows the mechanisms and protocols used for communication between them.

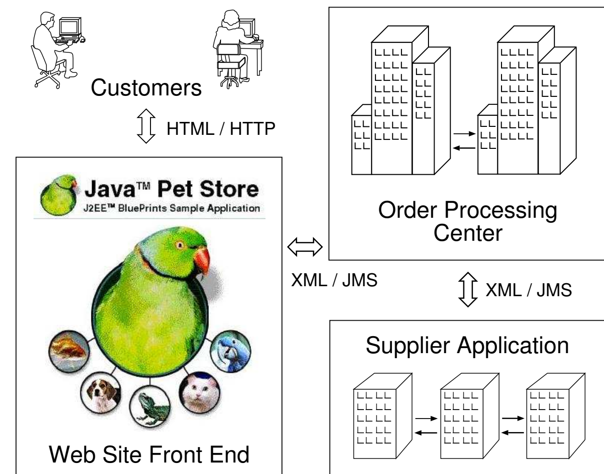

Figure 1: Main Pet Store Sub-Applications

2.2 Pet Store's Asynchronous, Document-Oriented Architecture

The Order Processing Center defines the business process for handling purchase orders placed at the Web site [SSJt02]. The business process consists of a workflow, which is a sequence of steps with transitions between them. Transitions between steps are handled by special classes called *transition delegates*, which allows for flexibility if the type of communication between steps is to change. Pet Store uses a document-oriented business process to coordinate its internal workflow and to communicate with

its supplier application. Transition delegates pass XML documents between workflow steps by placing JMS messages in message queues (or topics). The JMS queues are the transition points between steps. Messages arriving in the queues are processed asynchronously by Message-Driven Beans (MDBs). The asynchronous architecture allows components to call each other without having to block and wait for a response.

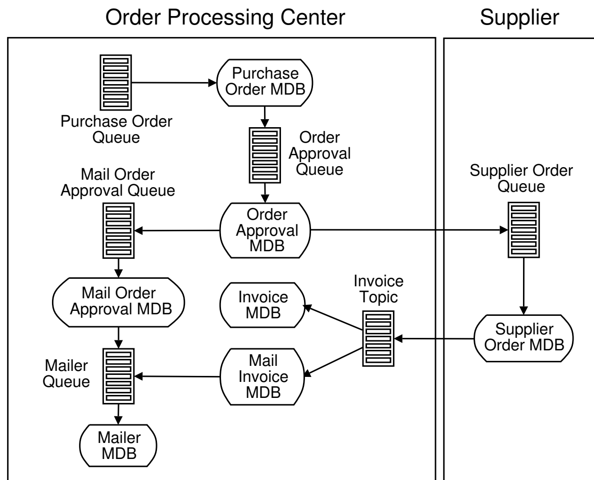

Figure 2: Message Flow in the OPC and Supplier [Sund]

Figure 2 depicts the flow of messages upon reception of a purchase order at the OPC. The OPC receives the order through its *purchase order queue*, validates it and passes it to the *order approval queue*. The order approval MDB confirms if funds are available for the order. Once the order is approved, it is sent to the supplier. The latter receives the order through its *supplier order queue*, validates it and ships products to the customer. It also creates an invoice and returns it to the OPC by sending a JMS message to the *invoice topic*. Two MDBs, *invoice* and *mail invoice*, both listen to the invoice topic and process the arrived invoice. The invoice MDB triggers the OPC's invoice processing workflow. The mail invoice MDB sends a message to the *mailer queue* in order to notify the customer by email that his order is completed.

2.3 Web Services in Pet Store

As mentioned earlier, starting with version 1.3.1, Pet Store includes Web Service interfaces to some of its components. In particular, Web Service interfaces to the supplier application are provided, allowing for the communication with it to take place through Web Services. This decouples the supplier from the OPC enabling for integration with external suppliers running on non-J2EE-based platforms.

Figure 3 shows the new message flow leveraging Web Services for the communication with the sup-

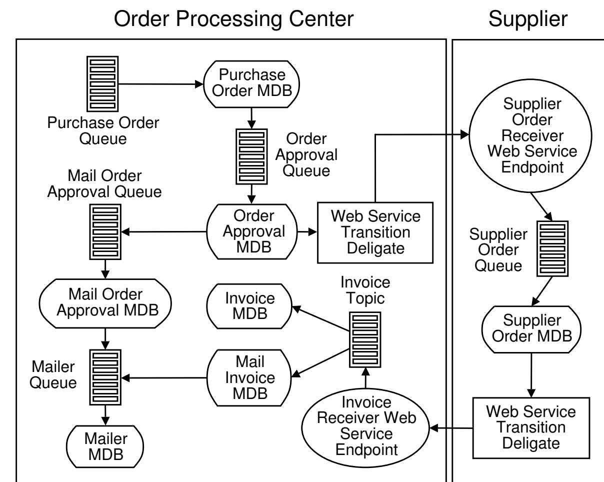

Figure 3: Message Flow when using Web Services[Sund]

plier. Instead of sending orders directly to the supplier through JMS, the OPC now sends them to a Web Service endpoint in the supplier using a modified transition delegate. The supplier Web Service endpoint receives orders as XML documents and validates them against a public XML schema. It then places them in the supplier order queue, which triggers the usual order processing workflow.

Similarly, instead of sending invoices directly to the OPC through JMS, the supplier now sends them through a Web Service endpoint in the OPC. The latter receives incoming invoices, validates them against a public XML schema, and places them in the invoice topic, triggering the usual invoice processing workflow.

The new Pet Store code can be compiled in two variants. In the first one (*JMS variant*), no Web Services are used and communication with the supplier is done by directly sending messages to JMS targets. In the second one (*Web Service variant*), Web Services are used as explained above.

2.4 Use of Transactions

Pet Store uses atomic transactions throughout its sub-applications to ensure data integrity and reliable order processing. The OPC and the supplier sub-application execute workflow steps in separate transactions (*workflow transactions*). Each workflow transaction has the reception of the process step's activating message as the first operation and continues with the business actions taken by the respective MDB. The transaction ends after the transition delegate has performed the necessary operations (usually sending of a JMS message) to activate the next workflow step. Any error during the execution of a processing step causes the workflow transaction to be rolled back and all changes to be undone. The initiating message remains in the queue available for further processing

attempts.

Messages sent to queues or topics for activation of workflow steps only become visible after the transactions from which they are sent commit. This is due to the isolation property of transactions and ensures that order processing steps are executed strictly sequentially. In the Web Service variant, however, the above behavior cannot be guaranteed for the workflow steps that are activated through Web Service calls (transition from OPC to supplier and back). While transactions are started in the same manner as in the JMS variant, the Web Service calls do not transport the transaction context across the OPC/supplier interface. As a result, actions taken at the Web Service endpoints do not participate in the calling transactions, which can be considered a functional weakness. While errors in executing the Web Service calls and failures during the actions at the Web Service endpoints are correctly propagated back to the caller causing it to abort the transaction, the converse is not true. Should an error occur in an action after a successful Web Service call or at commit time of the transaction, the necessary roll back cannot be propagated to the opposite side of the OPC/supplier interface.

Although the described correctness problem does not become manifest in our experiments and no errors occur during order processing, the lack of transaction context propagation still has a performance implication. Because Web Service calls across the OPC/supplier interface do not participate in any transactions, the isolation property is not maintained with respect to the visibility of sent messages. This allows supplier orders to arrive at the supplier business logic *prior to* completion of all business actions in the originating OPC processing steps. This does not pose a correctness problem per se under reasonable execution timings[1], however, it allows the supplier to begin processing submitted orders earlier. Since the transition between the OPC and supplier is part of the time critical path from order acceptance to completion, it affords a response time advantage for the Web Service variant.

To provide a level playing field for performance comparisons between the JMS and Web Service variants, the above problem had to be addressed. The preferred solution for the Web Service variant would be to adopt the transactional behavior of the JMS variant. Unfortunately, standards-based Web Service transaction technology is not yet fully developped. Currently, no widely supported standards for transactional Web Services exist and no products are available, yet. However, specifications to support transactions across Web Service bounderies are proposed and being evaluated (WS-Transaction/WS-Coordination [CCC+02, CCF+02], BTP [OAS02]). With the products currently available, this issue could only be resolved by modifying the JMS variant to behave like the Web Service variant. Starting from the JMS variant as originally published by the BluePrints program [Suna] (now also referred to as the *original JMS variant*), we made the the BluePrints program [Suna] (hereafter referred to as *original JMS variant*), we made the following changes. The *order approval transition delegate* was modified to send its order to the supplier application outside the running transaction. The same was done for the symmetrical case of the *supplier order transition delegate* sending its invoice back to the OPC application, so that both variants can take advantage of the early availability of messages sent across the OPC/supplier interface.

[1] The sending workflow transaction has to complete before the whole supplier business logic gets executed and sends the invoice back to the OPC application.

3 Experimental Setting

3.1 Test Scenario

In this section we take a look at the workload used as a basis for our experiments. Since our goal was to study the effect of using Web Services on Pet Store's overall performance, we chose a load profile which puts the stress on the back end, where Web Services come into play. To this end, we generate heavy purchasing activity by simulating goal-oriented customers. Each customer enters the store, selects a product from the catalog and then proceeds to checkout to order it. The customer is simulated using a client thread that goes through the steps shown in Figure 4. Each step corresponds to one HTTP request (the first 6 GET, the remaining POST) and the whole scenario corresponds to one user session. We used the *Siege* tool [Ful02] (version 2.57b2) to generate the load. The tool creates a client thread for each user simulated to repeatedly execute the scenario, putting a uniformly distributed delay of 1-4 seconds between requests. This should correspond to a simplified representation of the user's think time.

3.2 Deployment Environment

We deployed Pet Store in the deployment environment depicted in Figure 5, using Web Logic Server 7.0 (SP1) as a J2EE container and Oracle 9i (9.0.1) as a database server.

The database server is used for persisting application data (entity beans) and JMS messages. The client machine is used to generate the load (using the *Siege* tool) and control the progression of the experiments. Additionally, the client machine is running a mail server, used by the application server for sending Pet Store's email notifications to customers about the progress of their orders. Exim version 3.35 [Met03] was employed as a Message Transport Agent (MTA).

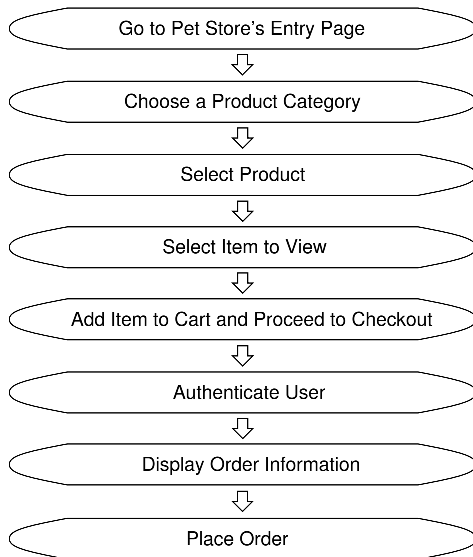

Figure 4: Test Scenario

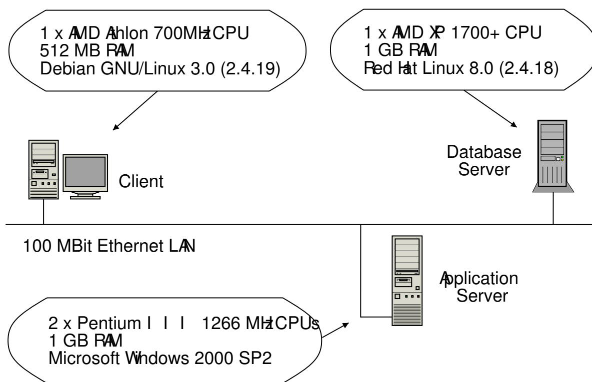

Figure 5: Deployment Environment

3.3 Measurements Taken

In order to quantify system performance under the considered workload, different measurements were taken during the experiments. For this purpose, Pet Store was instrumented to log timestamps at different points during the execution (*measuring points*) and performance metrics were calculated based on differences between timestamps.

The first measurement determines the start of an order's processing and is done in Pet Store's front end right after the reception of an order before it is forwarded to the OPC. The second measurement determines the end of an order's processing and is done in the Mailer MDB at the point an email is sent to the client to confirm order completion. The difference between timestamps taken at these points, gives us the total order processing time, also referred to as the *response time*. Dividing the number of orders processed during an experiment by the time elapsed between its start and end, we obtain the *throughput*.

To further monitor the behavior of the back end, three additional measurements were done. The first one determines the time at which the OPC calls the supplier to submit a new order, *i.e.* the beginning of the calling operation. When running in JMS mode this is the time of sending a JMS message to the supplier order queue. When running in Web Service mode, it is the time of calling the supplier Web Service endpoint. The second measurement determines the time at which the Web Service call (or the sending of the new order, respectively) completes. The difference between timestamps taken at these points, gives us the time for which the OPC has blocked when calling the supplier. We refer to this time as the *call time*.

The last measurement is done in the supplier order MDB and determines the time of arrival of an order as an XML document and the start of its processing. The time elapsed between the beginning of a call to submit an order to the supplier and the start of its processing will be referred to as the *latency*.

4 Performance Analysis

To gather the measurements for our analysis, we performed three series of experiments. Each series consists of six experiments characterized by the number of client threads used for simulating concurrent customers (15, 25, 40, 60, and 100 client threads, respectively). The first series represents the base case and uses the modified JMS mode (referred to as "JMS" in the following figures). Send operations at the OPC/supplier interface in this mode are not part of the workflow transactions as described in section 2.4). The second series covers the Web Service variant of Pet Store ("Web Service") .

Finally, the third series uses the original JMS variant of Pet Store as published by the BluePrints program, *i.e.* the JMS send operations at the OPC/supplier interface are part of their respective workflow transactions. These measurements represent the typical operational behavior of the Pet Store, but cannot be used for comparison with the Web Service variant as discussed in section 2.4. They are presented to highlight the performance differences of our base case system (JMS) with respect to the fully transactional, original JMS mode system.

The CPU utilizations on the client machine and database server machine were monitored during all the experiment runs and stayed below 10 and 35 percent, respectively, over the range of all simulated load levels.

4.1 Throughput

Figure 6 shows the throughput results (as the number of completed orders per minute) for each experiment in JMS and Web Service mode. The original JMS mode is not depicted as it exhibits identical throughput characteristics as the modified JMS mode (less than 1 percent deviation) over the full load range.

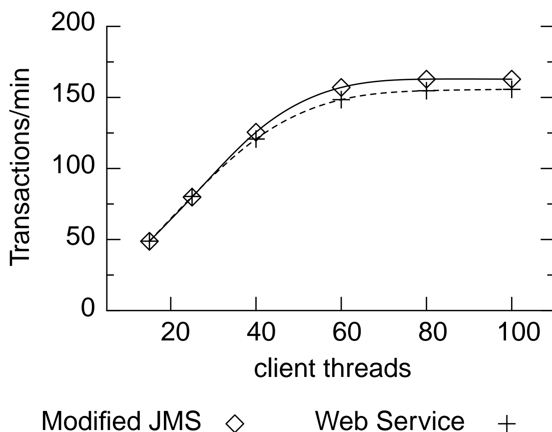

Figure 6: Throughput Measurements

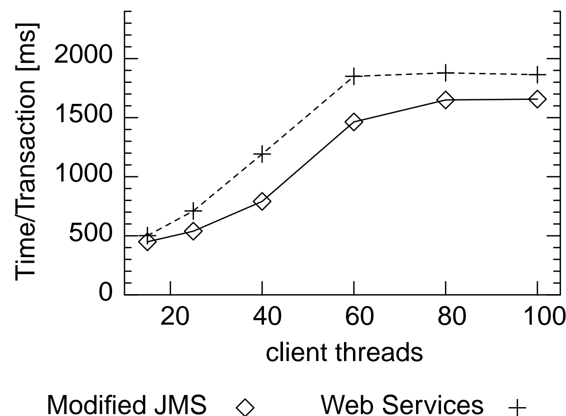

Figure 7: Response Time Meassurements

The JMS mode achieves the highest throughput of 163 orders per minute with 80 concurrent client threads. Under the same load, the Web Service variant reaches an about 5 percent lower value of 155 orders per minute. Under light to moderate load of up to 40 client threads, both variants scale linearly and exhibit almost identical performance. The differences remain below 1 percent in this load range, but we observe about 10 percent higher CPU time demand for the Web Service variant. This difference in resource consumption has no impact on throughput, as enough CPU time is readily available at that point. In the load range beyond 40 client threads towards system saturation, the higher CPU consumption begins to have an impact on throughput. The point of system saturation where no additional increase in throughput can be achieved is reached at a load of 80 client threads in the JMS case. Due to the higher CPU time demand of the Web Service variant, it reaches that point already at a load close to 70 threads and thus lags behind in maximum throughput.

4.2 Response Time

When considering the average total time for processing an order (Figure 7), it stands out that the response time does not grow beyond all limits under increasing load. The system remains in a stable state in which the number of orders injected into the system per unit of time equals the number of orders completed. This relates to an observation of the number of orders in the different stages of processing. The typical length of the queues involved in order processing did not grow beyond 2 messages even under extreme load. The Mail queue is the sole exception with a length of 5 messages due to the dispatching of three messages during the course of processing one order.

In addition to the well balanced breakdown of the order processing steps, the synchronous order injection has a stabilizing effect on the response time. The increase in the number of client threads, throughput and resource utilization causes the Pet Store front end to present a less responsive interface to the client. The increasing response times of HTTP requests to the front end slow down the scenario execution and thus the eventual injection of an order. This effect attenuates the injection rate increase of additional client threads such that the effective injection rate does not grow beyond the application server's capacity.

In contrast to throughput, the average response time exhibits more pronounced differences between the JMS and Web Service variant. The response time under light load (15 client threads) starts off with a 50 ms penalty (11 percent) for the Web Service mode at about 500 ms. The additional time spent in order processing under low system utilization can be attributed entirely to the higher OPC/supplier interface latency of the Web Service variant (see section 4.3).

The faster growing CPU utilization of the Web Service variant under increasing load causes the response time to increase faster as well. The higher CPU consumption for Web Service processing leaves less CPU time for other tasks, so that their processing time increases as well — a secondary effect of the use of Web Services. Together with the increased latency, this leads to the penalty for the Web Service variant reaching up to 50 percent at 40 client threads. As the load increases further to 60 client threads, the response time difference stays mostly constant at 400 ms due to the stabilization effect of the Pet Store setting in under heightened CPU utilization. The Web Service variant already reaches its capacity limit at that point and has a response time of 1850 ms, exhibiting a 25 percent increase over the JMS variant. The application server's CPU resources are saturated at that point, while the database server shows 35 percent CPU utilization. The JMS variant reaches that state at a load of 80 client threads, at which point the response time difference falls back to about 13 percent. The response time advantage

of the JMS variant melts off quickly in the 60 to 80 client threads load range as a result of the JMS variant no longer enjoying a non-saturated CPU while the Web Service variant already suffers from scarce CPU time at this load interval.

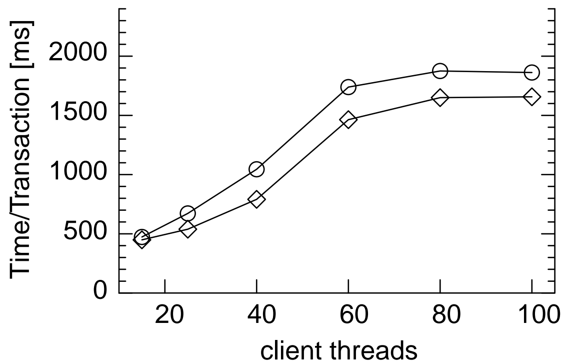

Figure 8: Response Time Meassurements

The comparison between the modified JMS variant and the original JMS variant (Figure 8) shows similar response time characteristics. The original JMS variant line runs parallel to the JMS variant at about 200-250 ms higher values except for light loads, where the difference amounts to 25 ms. The system monitor readings from the application server show that the CPU utilization is virtually identical for these variants. Reasons for the response time discrepancy can be found in the impact of the modifications on the OPC/supplier interface latency.

4.3 Latency and Call Time

While throughput and response time characterize the speed of order processing as a whole, the latency and call time are directly related to the OPC/supplier communication and are thus significantly affected by the introduction of Web Service interfaces. Figures 9 and 11 show the call time and latency measurement results of the modified and original JMS variants.

Considering the latency and call time of the modified JMS variant, it strikes that messages sent, arrive at their destinations (latency — Figure 11, diamond mark) *prior to* the sending entities having completed send operations (call time — Figure 9, diamond mark). The reasons for this become clear from looking at Figures 10, which shows the sequence of actions performed when sending JMS messages to the supplier.

The non-transacted send procedure can only return to the caller after the messaging system can guarantee that the message will be delivered at some point. The configuration at hand persists the message in a database to provide this guarantee without

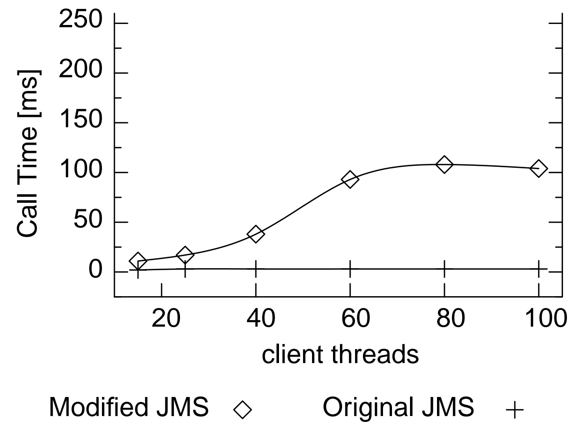

Figure 9: Call Time Measurements

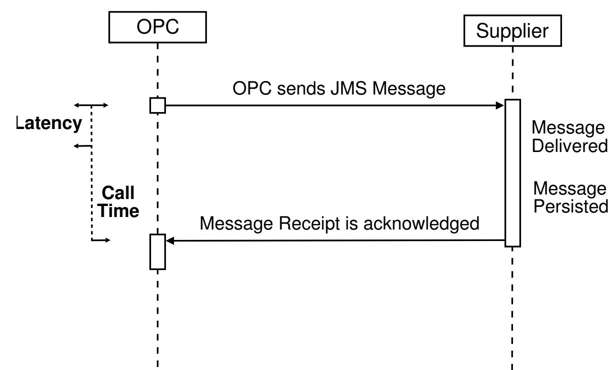

Figure 10: JMS Call in the Modified JMS Variant

waiting for an actual successful reception to complete. Should an error occur during the activities of the receiving supplier order MDB causing the transaction to roll back, the messaging system can always redeliver the message since it can be retrieved from persistent storage. The call time for the modified JMS variant starts at 11 ms under light load and increases slowly until it stabilizes at values around 100 ms under high load (similar to throughput). These characteristics are related to the additional database activity for persisting messages due to increasing throughput as well as the growing time needed to serialize the message for persistence due to increasing CPU utilization.

The first delivery of the message to the supplier order MDB does not, however, have to wait for the completion of the operations to persist the message, but can start immediately. This allows the modified JMS variant to sustain a very low latency of 2 to 4 ms over the whole load spectrum. This low latency of the OPC/supplier interface has a significant impact on the overall response time of the modified JMS variant, as it determines the time the next processing step can begin and as such is part of the time-critical path.

Comparing these results to the original JMS variant (cross mark) reveals very different call time and latency behavior for the latter. While the sent message

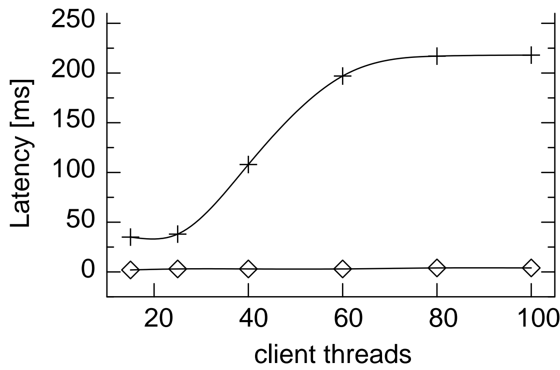

Figure 11: Latency Measurements

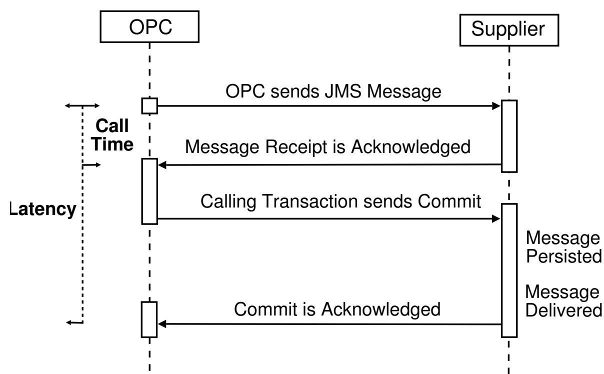

Figure 12: JMS Call in the Original JMS Variant

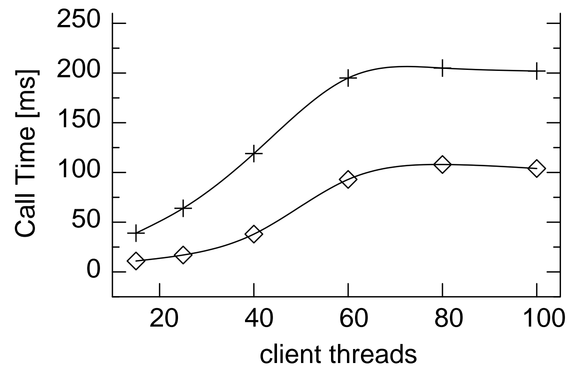

Figure 13: Call Time Measurement Comparison

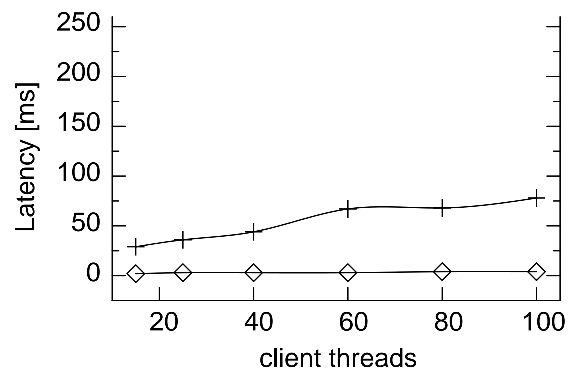

Figure 14: Latency Measurement Comparison

has to be persisted before the send method call can return in the modified JMS variant, the send operation being part of a transaction in the original JMS variant can defer persistence to the later commit operation. This causes the call time to stay very low at 2-3 ms throughout the full load range. On the other hand, the availability of the sent message to the receiving end at the supplier application is delayed until after the completion of the transaction in the commit operation. This prolongs the latency not only by the time needed to persist the message (as measured by the call time readings for the non-transacted send in the modified JMS variant), but also by the time needed to commit the work of all other actions taken in the business logic of this processing step. This prolonging effect of measured call time in the modified JMS variant on latency in a transacted variant should be kept in mind when considering the measurements of the following comparisons to the Web Service variant.

Figures 13 and 14 depict the call time and latency measurements for the Web Service variant in comparison to the modified JMS mode experiments. As both variants under consideration are performing the send operation for invoking the supplier subapplication outside the transaction of the workflow step, their latencies are not burdened by the time needed to achieve message persistence. For the same reason, both variants allow immediate visibility of the sent message as they are not bound to any isolation requirements. Together, this makes their latency and call time results comparable.

While the modified JMS variant exhibits a 2-4 ms latency as noted above, the Web Service variant shows a noticeably higher latency. Under light to moderate load (15 to 40 client threads range) of linear scalability, equal throughput and unsaturated CPU resources, the latency increases slowly from 29 ms to 44 ms. With CPU saturation setting in, the latency reaches 67-78 ms in the high load and overload range. As part of the latency time, a SOAP message for the Web Service call gets created, serialized and transmitted to the Web Service endpoint of the supplier application over the (host internal) network. The receiving end in the application server parses the SOAP message and invokes the implementation method for the Web Service operation with the supplier order XML document string as argument. The document is sent to the supplier order queue and finally received at the supplier order MDB. As seen with the modified JMS variant, the last two steps (send-

ing and receiving the message) together take 2-4 ms to be executed. The remaining 27-74 ms of latency account for performing the described steps of the Web Service call.

The call time of the Web Service variant consists mainly of three components. As the Web Service operation gets called synchronously, the caller has to block until reception of the reply. As a first component, it takes the above mentioned 27-74 ms for the call to arrive at the Web Service implementation method. The call time to send the passed argument string (supplier order XML document) to the supplier order queue follows. Since this is a non-transacted send operation under identical conditions as in the modified JMS method, the call time shares the same characteristics as the call time of that variant (Figure 9, diamond mark) and decisively shapes the call time curve. The remaining time component is used to create, transmit and interpret the reply message of the Web Service call.

While the call time itself has no impact on the overall response time in the variants using non-transacted send operations, it nonetheless puts the latency readings into perspective. As seen in the comparison between the modified and original JMS variants, the call time measurements can give an indication for potential latency implication once transactions are supported and used in the Web Service variant.

4.4 Effects on Garbage Collection

The *BEA WebLogic JRockit* Java Virtual Machine (JVM) version 7.0 SP2 [BEAa] used in our experiments offers a number of garbage collectors. The *generational concurrent* garbage collector selected for our experiments[2] works as follows [BEAb]. New Objects are allocated in the young generation. When the young generation (called a nursery) is full, the JVM "stops-the-world" and moves the objects that are still live in the young generation to the old generation. An old collector thread runs in the background all the time; it marks objects in the old space as live and removes the dead objects, returning them to the JVM as free space.

The WebLogic JRockit JVM offers options for gathering and displaying statistical information on the garbage collection activities similar to the functionality found in the Sun JVM [Sunc]. The output of the JVM offers insight into the time and duration of each garbage collection run performed, as well as the total time spent for all garbage collections on the young and old generation over the runtime of the JVM. Figure 15 shows the garbage collection time for the modified JMS and Web Service variant experiment series.

[2]The garbage collector options `-Xms500m -Xmx500m -Xns40m -Xgc:gencon` led to good performance results in our environment and were used for all experiments. See [BEAb] for a description of all supported options.

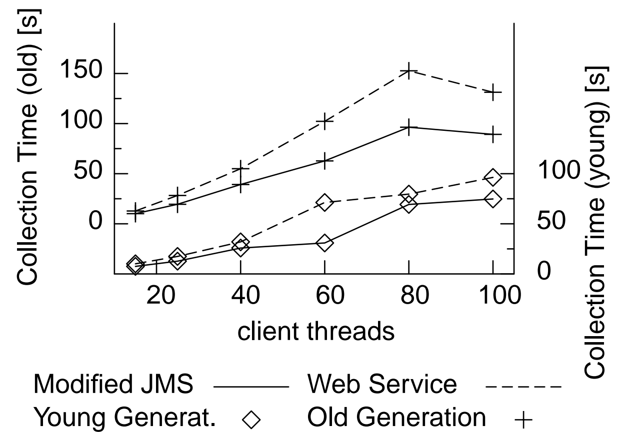

Figure 15: Garbage Collection Measurements

The increasing time needed for performing the garbage collection activities under growing load is an indication for a higher object throughput under load. The amount of object memory allocated and freed during the course of an experiment increases with the order throughput of the system. Our measurements show that the total garbage collection time of the Web Service variant significantly exceeds the one of the modified JMS variant, which indicates a higher level of object throughput in the former. The time spent in garbage collecting the young generation increases by 30 percent, while it is 40-50 percent higher for the old generation. The divergence of the young collection time at the 60 client threads data point seems to be an anomaly of the exact parameter values for the chosen garbage collector.

The garbage collection process of the young generation involving the transfer of all active (live) objects at that time to the old generation space needs to suspend all user threads to perform its task (so called "stop-the-world" method). This interruption of execution has a direct impact on the response times of all orders being processed at that instance in time. Based on the observation of the distribution of young collection occurrences and their duration in conjunction with the time an order is processed on the application server (response time), it can be derived that about every tenth order in the modified JMS variant is affected by an interruption of 35 ms on average under light load. Due to the quickly increasing number of young garbage collections and increase in response time under medium to high load (40-60 client threads) at the same time, a third of the orders are hit by a 40 ms execution interruption in that load range. Finally under extreme load of 80 and more client threads, the processing of almost every order gets suspended for about 80 ms due to garbage collection of the young generation. The distribution of young generation garbage collections in the Web Service variant has very similar characteristics to the one

of the JMS variant. However, the average duration of a collection run is about 30 percent higher and so directly contributes up to 25 ms to the response time penalty of the Web Service variant.

While the garbage collection of the young generation has a direct impact on response time due to the suspension of user thread execution, the old garbage collection acts on performance through a different mechanism. As the collection of the old generation is performed mostly without the interruption of user threads but rather concurrently with them, it is not affecting directly the response time or latency of individual orders. It rather takes effect on system performance by making the CPU resources more scarce for processing of the genuine workload. Up to 5 percent of the run time of the modified JMS variant is spent for garbage collection of the old generation; up to 8.5 percent for the Web Service variant. This drain on CPU time prolongs the execution of CPU intensive parts of the order processing accordingly.

5 Summary and Conclusions

In this paper we studied the effects on performance when part of the Java Pet Store application is implemented using Web Service interfaces. The reference implementations were the original Pet Store application with transactional messaging and a modified version of Pet Store with non-transactional, but still reliable messaging. The latter modification was needed for a fair comparison since today's Web Services Platforms do not provide support for transactional calls. The measurement results of our experiments show that the system throughput penalty from using Web Services in the Java Pet Store is only marginal. Under light to moderate load, the throughput decreases by less than 1 percent and falls behind the modified JMS variant by at most 5 percent under high and extreme load. The throughput difference stems from the Web Service variant reaching the threshold of CPU capacity under slightly lighter load due to this variant's higher CPU consumption.

The changes in response time behavior are less uniform. While the response time of the Web Service variant increases by about 20 percent under light and extreme load, the difference grows up to 50 percent in the moderate to high load range. In addition to the already mentioned higher CPU consumption, the typical 20 percent increase results from the higher latency of the transition between the sub-applications across the Web Service interface.

In addition to the direct costs of performing Web Service calls due to operations like serialization, transmission and parsing of SOAP messages, we could observe increased garbage collection activity in the JVM of the application server. The former operations led to higher memory allocation and deallocation rates for the Web Service variant. The Web Service variant of the Pet Store exhibits 30 percent longer execution interruptions for performing garbage collection on the young generation, which has a direct impact on the response time. An additional negative effect results from a 40 percent higher CPU resource consumption of the garbage collector in the Web Service variant. This amounts to a total of up to 8.5 percent of the available CPU time being spent on garbage collection.

Our measurements suggest that for non-transactional execution the performance penalty of using Web Services is minimal with respect to throughput, but up to 50% for response time under moderate to heavy load. Web Services are a viable implementation alternative for applications that do not require transactional behavior. We are currently working on transaction models for Web Services and are closely tracking the emerging implementations. We will report on the performance of transactional Web Services once stable implementations are available.

Acknowledgments

We would like to thank BEA Systems Inc. for providing the required licenses and support for conducting our reseach with the BEA WebLogic Application Server and the BEA JRockit JVM. The trademarks or registered trademarks used in this paper belong to their respective owners.

References

[BCF+03] David Booth, Michael Champion, Chris Ferris, Francis McCabe, Eric Newcomer, and David Orchardand. Web services architecture, 2003. W3C Working Draft 14 May 2003.

[BEAa] BEA Systems, Inc. BEA WebLogic JRockit. http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/jrockit/.

[BEAb] BEA Systems, Inc. Tuning the WebLogic JRockit 7.0 JVM Memory Management System. http://edocs.bea.com/wljrockit/docs70/tuning/memmgmt.html.

[CCC+01] Patrick Cauldwell, Rajesh Chawla, Vivek Chopra, Gary Damschen, Chris Dix, Tony Hong andFrancis Norton, Uche Ogbuji, Glenn Olander, Mark A Richman, Kristy Saunders, and Zoran Zaev. *Professional XML Web Services*. Wrox Press, 2001.

[CCC+02] Felipe Cabrera, George Copeland, Bill Cox, Tom Freund, Johannes Klein, Tony Storey, and Satish Thatte. Web services transaction (ws-transaction), August 2002. http://www.ibm.com/developerworks/library/ws-transpec/.

[CCF+02] Felipe Cabrera, George Copeland, Tom Freund, Johannes Klein, David Langworthy, David Orchard, John Shewchuk, and Tony Storey. Web services coordination (ws-coordination), August 2002. http://www.ibm.com/developerworks/library/ws-coor/.

[Cer02] Ethan Cerami. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*.

O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, 2002.

[Ful02] Jeffrey Fulmer. Siege — An Open Source Stress Tester, 2002. http://www.joedog.org/siege/index.shtml.

[Jus03] Kai S. Juse. Performance of J2EE-based Web Services. Master thesis, Darmstadt University of Technology, June 2003. In German.

[Met03] Nigel Metheringham. The Exim Message Transfer Agent, 2003. http://www.exim.org/.

[OAS02] OASIS Business Transactions Technical Committee. Business Transaction Protocol — Version 1.0, June 2002. http://www.oasis-open.org/committees/business-transactions/documents/specification/2002-06-03.BTP_cttee_spec_1.0.pdf.

[SSJt02] Inderjeet Singh, Beth Stearns, Mark Johnson, and the Enterprise Team. *Designing Enterprise Applications with the J2EE Platform*, chapter 11: Architecture of the Sample Application. Addison-Wesley Pub Co, 2nd edition, 2002. http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch3.html.

[Suna] Sun Microsystems, Inc. Java BluePrints - Guidelines, Patterns, and Code for end-to-end Java Applications. http://java.sun.com/blueprints/.

[Sunb] Sun Microsystems, Inc. Java Blueprints Enterprise Patterns. http://java.sun.com/blueprints/patterns/index.html.

[Sunc] Sun Microsystems, Inc. Java HotSpot Technology. http://java.sun.com/products/hotspot/.

[Sund] Sun Microsystems, Inc. Java Pet Store Sample Application. Documentation. http://java.sun.com/blueprints/code/jps131/docs/index.html.

[Tid00] Doug Tidwell. Web services: the Web's next revolution. IBM developerWorks — Web service, November 2000. http://www-106.ibm.com/developerworks/webservices/edu/ws-dw-wsbasics-i.html.