On the Performance of Transactional Remote Invocations over Commonly Used Transports

Ivan Silva Neto   Francisco Reverbel

Department of Computer Science
University of São Paulo
{ivanneto,reverbel}@ime.usp.br

Abstract

We have measured the performance of transactional remote invocations over three commonly used transports: IIOP, SOAP/HTTP, and JBoss Remoting. In the IIOP case, our transactional invocations followed the CORBA OTS standard. In the SOAP/HTTP case, they followed the WS-Coordination and WS-AtomicTransaction standards. In the (non-standard) JBoss Remoting transport, they employed a transactional layer modeled after CORBA OTS. For each of those three transports, we evaluate the overhead of propagating the transactional context, the cost of creating and committing empty transactions, and the total overhead incurred when the two-phase commit protocol actually runs. We also evaluate the impact of the transaction log on the total overhead.

Index Terms: *Transactions, performance, transactional invocations, CORBA OTS, WS-AtomicTransaction.*

1. Introduction

The ability of grouping two or more service requests into an atomic unit is a crucial feature of contemporary middleware platforms. Atomicity across remote requests is typically achieved by adding transactional capabilities to a remote method invocation (RMI) mechanism. Two standard transports are currently used for remote method invocations: IIOP is the RMI protocol defined by CORBA and adopted by Java EE; SOAP/HTTP plays a similar role in the Web services architecture. Even though it standardizes a transport (IIOP) for remote invocations, Java EE allows application servers to employ other transports as well. As a result, a number of proprietary protocols are also used in Java EE environments.

Two current standards address transactional remote invocations. The Object Transaction Service (OTS) specification [16] adds transactional capabilities to the remote invocation mechanism provided by CORBA. A pair of WS specifications, WS-Coordination (WS-C) [15] and WS-AtomicTransaction (WS-AT) [14], does the same for the

SOAP/HTTP stack. Similar "transactional layers" can be defined over non-standard RMI mechanisms. Each such layer comprises:

- a service for creating and committing (or aborting) global transactions;
- the definition of a data structure — the transactional context — that must be propagated along with each remote invocation performed as part of some transaction;
- a remotely accessible `Coordinator` interface, which supports dynamic registration of transaction participants ("resources") with the transaction coordinator;
- a remotely accessible `Resource` interface, through which the coordinator drives the two-phase commit (2PC) protocol at transaction completion time.

Creating a global transaction means assigning a global id to the transaction and creating a `Coordinator` for the transaction. The global transaction id and a remote reference to the `Coordinator` are the most important fields of the transactional context carried along with service requests. As a result of the propagation of that reference, the remote servers involved in a distributed transaction can reach the `Coordinator` in order to register transactional `Resources` as 2PC participants. Note that the propagated `Coordinator` reference (and hence the transactional context) may take different forms, depending upon the RMI transport employed: it can be a CORBA object reference, a Web service endpoint reference, or some similar artifact.

The addition of a transactional layer atop an RMI mechanism certainly takes a toll in performance. This paper offers an experimental evaluation of the costs inherent to such layers. It presents the results of our experiments with transactional remote invocations in a Java EE scenario. We have used a popular Java EE server, the JBoss Application Server (JBossAS) [8], and three RMI transports supported by that server: IIOP, SOAP/HTTP, and the non-standard protocol implemented by the JBoss Remoting framework [11]. We have included the JBoss Remoting transport in this study for the following reasons: *(i)* it is the RMI protocol that JBossAS employs by default; *(ii)* as such, and given the wide usage of JBossAS, it is a very commonly

used transport; *(iii)* it can be loosely regarded as a representative of the proprietary transports implemented by various Java EE servers.

Given the central role of distributed transactions in enterprise computing, the scarcity of published results on the performance of transactional remote invocations is a very surprising fact. How much does it cost to propagate the transactional context along with remote invocations? Is the round-trip time of a transactional remote invocation comparable to the one of a plain (non-transactional) remote invocation? What is the cost of starting and ending global transactions? How big is the overhead of the 2PC protocol? How much of this overhead is due to the 2PC logging activity, which involves synchronous (disk-forced) writes of the relevant 2PC events to stable storage? If we look at remote invocations over different transports, how do all the "transactional costs" vary across transports? To the best of our knowledge, these questions have not been answered before.

Even though JBossAS supports multiple RMI transports, its default transaction manager currently offers full support only to the IIOP transport. For that reason, we have configured JBossAS to use another transaction manager. In the next section of this paper, we summarize the features of our chosen transaction manager, XActor [17], which fully supports distributed transactions over IIOP, SOAP/HTTP, and JBoss Remoting. Section 2 also has further details on the advantages of using XActor instead of the transaction manager included in JBossAS. Section 3 describes the hardware and software environment in which we ran all our experiments. The core of this paper is Section 4, which offers answers to the questions posed in the preceding paragraph. While the numbers in Section 4 are specific to our JBossAS/XActor scenario, they convey qualitative information that is likely to be relevant to other transactional scenarios as well. The last two sections review related work (Section 5) and present our concluding remarks (Section 6).

2. XActor

XActor[1] is a distributed transaction manager written in Java and aimed at server-side application containers such as Java EE servers and dependency injection frameworks. A running XActor instance is typically associated with an application container instance and exists in the same server process as that container. XActor fully supports failure recovery: it performs write-ahead logging of the relevant 2PC events and implements a recovery procedure that runs after server crashes.

A plug-in architecture allows XActor to coordinate distributed transactions over an open-ended set of RMI mechanisms and transports. Each XActor plug-in is a dynamically

deployable module that is specific to a given RMI mechanism. It cooperates with XActor to extend that mechanism with transactional capabilities.

Three XActor plug-ins are currently available: an IIOP plug-in, which implements the CORBA OTS standard, a SOAP/HTTP plug-in, which implements the WS-C and WS-AT specifications, and a JBoss Remoting plug-in, which implements a non-standard transactional layer modeled after CORBA OTS. We have used these plug-ins in our experiments.

In comparison with the transaction manager included in JBossAS, XActor offers a broader support to distributed transactions over multiple transports. The transaction manager shipped with JBossAS also implements distributed transactions over SOAP/HTTP, but at this time it does not provide a remotely accessible interface for transaction demarcation by SOAP clients, nor does it support failure recovery for Web service transactions. Moreover, support to distributed transactions over JBoss Remoting is not yet available in that transaction manager.

3. Experimental environment

3.1. Hardware

In our experiments, we used a set of P4 3.0 GHz machines equipped with 1 GB of RAM and a 7,200 RPM disk. The machines were connected through a switched Ethernet LAN of 100 Mbps.

3.2. Software

All the machines ran the Linux operating system, version 2.6.17. Sun's Java SE Runtime Environment version 5.0 (1.5.0_01-b08) was installed on those machines, as well as JBossAS version 5.0.0.Beta2. More precisely, the JBossAS version we used in the experiments comprised the following components: JBoss Remoting 2.2.0.GA, JBoss Serialization [12] 1.0.3.GA, JacORB [3] 2.2.4.jboss.patch1, JBossWS [13] 2.0.0.GA, and Woodstox [21] 3.1.1.

We have made five adjustments to the out-of-the-box configuration of JBossAS. First, we disabled the trace log, used to record information about the activities performed by the application server. If that log were left enabled, a considerable amount of information would be recorded during the execution of a distributed transaction, thus increasing the average execution time of the transaction. Second, we configured the JBoss Remoting framework to use JBoss Serialization instead of plain Java serialization. This change allowed JBoss Remoting to achieve its best performance[2]. Third, we set the Java heap size to 768 MB, using the `-Xms768m` and `-Xmx768m` parameters of the virtual machine. By allocating as much memory as possible to

---

[1]Project website: `http://xactor.sourceforge.net/`. All the source code for XActor is available as free software at the project website.

[2]No similar option exists for IIOP and SOAP/HTTP.

the Java heap, we minimized the number of garbage collections that took place during our experiments. Fourth, we replaced the database engine included in JBossAS with Apache Derby [2] 10.1.3.1, a database management system (DBMS) with support to transactions. Finally, the fifth and most significant adjustment was the replacement of the default transaction manager with XActor.

4. Performance experiments

4.1. Methodology

Each performance experiment comprised a sequence of $N_1 + N_2$ iterations, logically divided in two phases: the *warm-up phase* consisted of the first N_1 iterations and had the purpose of bringing the system to a steady state; the *measurement phase* consisted of the remaining N_2 iterations, whose times were measured. The charts and statistics in this section summarize the data collected in the measurement phases of our experiments.

In the first and second experiments, each iteration was a single remote invocation. In the third experiment, each iteration was a pair of remote invocations. In the fourth (and last) experiment, each iteration was a sequence of up to four remote invocations. We used $N_1 = 5,000$ and $N_2 = 15,000$ in the first three experiments, and $N_1 = 2,000$ and $N_2 = 8,000$ in the fourth experiment. In all the experiments, the client and server(s) ran on distinct machines. We performed all the measurements at the client side, using the `System.nanoTime()` method of the standard Java library.

4.2. Non-transactional remote invocations

The main goal of our first experiment was to provide a yardstick against which transactional remote invocations could be evaluated. We have measured the round-trip times of plain (non-transactional) remote invocations over IIOP, SOAP/HTTP, and JBoss Remoting. The invocations were targeted at a stateless EJB component [19] that was simultaneously accessible via those three transports and had its transaction attribute set to `Never`.

For each transport, an EJB client invoked the following methods:

- `getInt100()`, which returns an array containing 100 integers;
- `getDouble100()`, which returns an array containing 100 doubles;
- `getString100()`, which returns a string of length 100;
- `getString1000()`, which returns a string of length 1,000;
- `getString10000()`, which returns a string of length 10,000;
- `getDataSet()`, which returns an instance of the `DataSet` class, shown in Fig. 1;

```
public class DataSet implements java.io.Serializable {
   private byte _byte;
   private boolean _boolean;
   private short _short;
   private int _int;
   private long _long;
   private float _float;
   private double _double;
   private String _string;
   ... // Getters and setters.
}
```

**Figure 1. The `DataSet` class.**

- `getDataSet10()`, which returns an array containing 10 `DataSet` instances;
- `getDataSet100()`, which returns an array containing 100 `DataSet` instances.

Each method had a trivial implementation that just returned a constant value. Therefore, most of the time spent in a remote invocation was due to marshalling and communication over the selected transport (IIOP, SOAP/HTTP, or JBoss Remoting). We exercised a variety of return types to determine their impact in the duration of the remote invocations.

Experimental results. Fig. 2 summarizes the results of our first experiment. More specifically, the bar chart in Fig. 2a depicts the average round-trip times of the remote invocations, while the table in Fig. 2b presents the same round-trip times, plus information on the total number of bytes exchanged per remote invocation (the sum of the sizes of the request and response messages).

Fig. 2 shows that IIOP and JBoss Remoting perform quite similarly for non-transactional remote invocations. More accurately, JBoss Remoting is on average slightly faster than IIOP. SOAP/HTTP, however, is considerably slower than those two. The asymmetry between SOAP and the other two transports was expected, as SOAP is a textual, XML-based, transport, while IIOP and JBoss Remoting are both binary transports. The performance gap is most notable for the `getDataSet100` method, where SOAP/HTTP is about 20 times slower than IIOP or JBoss Remoting.

The results show that the influence of the return types on the round-trip times is larger in the case of SOAP/HTTP. For this transport, the methods that returned arrays or complex data types were considerably slower than the ones involving just character strings. For example, a SOAP call to `getDataSet10` took on average 9.29 ms, while a SOAP call to `getString10000` took only 6.57 ms on average (29% faster). Note that the SOAP invocation of `getString10000` was faster than the `getDataSet10` invocation, even though it involved bigger messages (11,010 bytes exchanged per `getString10000` invocation, against 2,788 bytes exchanged per `getDataSet10` invocation). This fact shows that the number of bytes exchanged is not
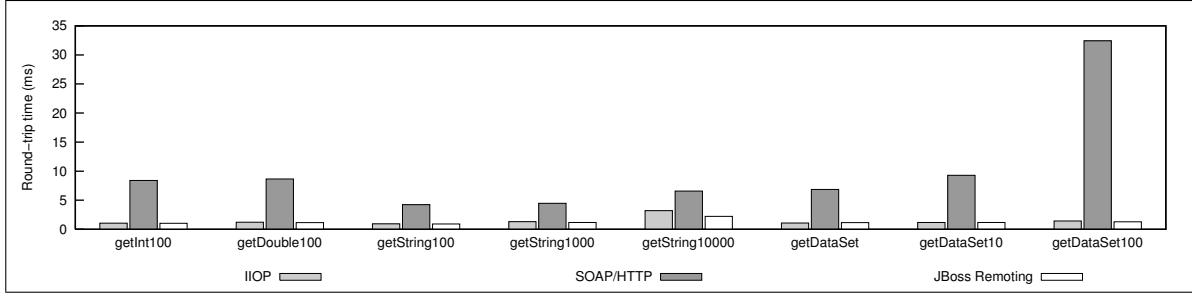
(a) Average round-trip times.

		getInt100	getDouble100	getString100	getString1000	getString10000	getDataSet	getDataSet10	getDataSet100
IIOP	Average round-trip time	1.05 ms	1.21 ms	0.93 ms	1.30 ms	3.20 ms	1.07 ms	1.17 ms	1.42 ms
	Data transfer per invocation	548 bytes	956 bytes	324 bytes	2,124 bytes	20,128 bytes	304 bytes	480 bytes	1204 bytes
SOAP/HTTP	Average round-trip time	8.40 ms	8.65 ms	4.23 ms	4.48 ms	6.57 ms	6.85 ms	9.29 ms	32.43 ms
	Data transfer per invocation	2,938 bytes	3,190 bytes	1,096 bytes	2,000 bytes	11,010 bytes	1,190 bytes	2,788 bytes	18,776 bytes
JBoss Remoting	Average round-trip time	1.02 ms	1.15 ms	0.92 ms	1.16 ms	2.23 ms	1.15 ms	1.18 ms	1.28 ms
	Data transfer per invocation	1,315 bytes	1,715 bytes	995 bytes	1,895 bytes	10,895 bytes	1,142 bytes	1,259 bytes	1,709 bytes

(b) Detailed results.

Figure 2. Results of the first experiment.

the major component of the remote invocation cost.

In comparison with SOAP messages containing just strings, the ones that contain arrays or complex data types have much more XML elements and therefore demand considerable more work from the XML parser. For this reason, the marshalling/unmarshalling of those messages is slower. In other words, the more complex a SOAP message is, the more work it will demand from the XML parser, and slower the remote invocation will be.

Our results suggest that XML parsing is the bottleneck of the SOAP/HTTP transport. Fig. 2b shows that SOAP/HTTP and JBoss Remoting invocations to the method `getString1000` involve messages of approximately the same size (2,000 and 1,895 bytes, respectively). In spite of that, Fig. 2a shows that the JBoss Remoting invocation is much faster than its SOAP/HTTP counterpart (1.16 versus 4.48 ms, respectively).

SOAP has been frequently criticized for its verbosity. Nevertheless, our measurements show that one cannot blame just the verbosity of SOAP for the bad performance of that protocol. In some cases the other transports can be as verbose as SOAP, but are still much faster, as they do not incur the cost of XML parsing. In our experimental scenario, the (XML-based) marshalling/unmarshalling mechanism used by SOAP was much less efficient than the ones employed by JBoss Remoting (JBoss Serialization) and by IIOP. In other words, the SOAP/HTTP invocations were slower than those performed over IIOP or JBoss Remoting due to the parsing of XML messages. Previous research had already pinpointed XML parsing as the bottleneck of the SOAP protocol [1, 10, 4].

4.3. Transactional remote invocations

When a transaction spans more than one process, the transactional context needs to be propagated across the processes involved in the transaction. For example, if an EJB client performs a remote invocation — within the scope of a transaction — on an EJB deployed on some application server, the remote invocation must carry the transactional context. The EJB client could be either a stand-alone Java application or a component deployed in another server.

The goal of our second experiment was to evaluate the cost of context propagation. An EJB client performed a series of remote invocations against the same EJB component used in the previous experiment. However, this time we set to `Required` the transaction attribute of the EJB. Moreover, the EJB client performed the whole sequence of invocations within the scope of a single transaction. A transactional context was therefore propagated along with every invocation.

Experimental results. Fig. 3a shows the average round-trip times of the transactional remote invocations, emphasizing (in dark gray) the overhead due to the propagation of the transactional context. Except for the parts in dark gray, Fig. 3a is identical to Fig. 2a. The measured overhead does not include the time needed to create and commit the transaction; it corresponds only to the propagation of the transactional context. Fig. 3b presents the detailed results.

As we can see in Fig. 3a, the overhead imposed by the propagation of the transactional context is approximately constant for a given transport, i.e., it does not depend on the method being invoked. This is expected, since the transactional context has a roughly constant size for a given transport (about 760 bytes for IIOP, 730 for SOAP/HTTP, and 1,050 for JBoss Remoting), and the process of injecting, propagating, and extracting the context is identical for all the invocations performed over the same transport.

Fig. 3b also shows the average overheads caused by the propagation of the transactional context. In the case of IIOP, those overheads were close to 1.5 ms. For SOAP/HTTP and JBoss Remoting, they were around 1.0 and 0.4 ms, respec-
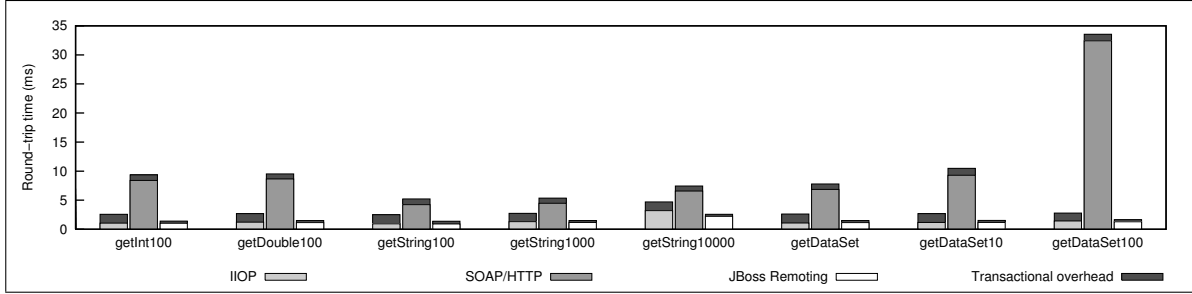
(a) Average round-trip times.

		getInt100	getDouble100	getString100	getString1000	getString10000	getDataSet	getDataSet10	getDataSet100
IIOP	Average round-trip time	2.58 ms	2.69 ms	2.50 ms	2.73 ms	4.70 ms	2.63 ms	2.68 ms	2.78 ms
	Data transfer per invocation	1,308 bytes	1,716 bytes	1,084 bytes	2,884 bytes	20,888 bytes	1,064 bytes	1,240 bytes	1,964 bytes
	Average overhead	1.53 ms	1.48 ms	1.57 ms	1.43 ms	1.50 ms	1.56 ms	1.51 ms	1.36 ms
	Transactional context size	760 bytes	760 bytes	760 bytes	760 bytes	760 bytes	760 bytes	760 bytes	760 bytes
SOAP/HTTP	Average round-trip time	9.35 ms	9.52 ms	5.21 ms	5.36 ms	7.44 ms	7.78 ms	10.48 ms	33.55 ms
	Data transfer per invocation	3,673 bytes	3,925 bytes	1,831 bytes	2,735 bytes	11,745 bytes	1,925 bytes	3,523 bytes	19,511 bytes
	Average overhead	0.95 ms	0.87 ms	0.98 ms	0.88 ms	0.87 ms	0.93 ms	1.19 ms	1.12 ms
	Transactional context size	735 bytes	735 bytes	735 bytes	735 bytes	735 bytes	735 bytes	735 bytes	735 bytes
JBoss Remoting	Average round-trip time	1.40 ms	1.49 ms	1.38 ms	1.50 ms	2.58 ms	1.49 ms	1.51 ms	1.64 ms
	Data transfer per invocation	2,367 bytes	2,767 bytes	2,047 bytes	2,947 bytes	11,947 bytes	2,194 bytes	2,311 bytes	2,761 bytes
	Average overhead	0.38 ms	0.34 ms	0.46 ms	0.34 ms	0.35 ms	0.34 ms	0.33 ms	0.36 ms
	Transactional context size	1,052 bytes	1,052 bytes	1,052 bytes	1,052 bytes	1,052 bytes	1,052 bytes	1,052 bytes	1,052 bytes

(b) Detailed results.

Figure 3. Results of the second experiment.

tively. It is surprising that IIOP — a binary transport — presented the largest overheads. After careful analysis, we found out that the main cause of those overheads was the marshalling and unmarshalling of CORBA references. The transactional context defined by the CORBA OTS specification contains two such references[3]. Those references have to be marshalled when the context is injected into an outgoing CORBA invocation, and unmarshalled when the invocation reaches its target. Our experiments have shown that the marshalling/unmarshalling of CORBA references is a slow process, at least in the case of the particular CORBA implementation (JacORB) used by JBossAS.

Even though IIOP presented a large overhead in our transactional scenario, it is still considerably faster than SOAP/HTTP (whose performance was again hurt by XML parsing). JBoss Remoting, in turn, is now noticeably faster than IIOP, mainly because the overhead of propagating transactional contexts over that transport is much smaller.

4.4. Transaction demarcation

The goal of our third experiment was to estimate the transaction demarcation cost, i.e., the time spent starting and committing empty transactions over IIOP, SOAP/HTTP, and JBoss Remoting. For each of these transports, a remote client started new transactions and immediately committed them. This experiment allowed us to measure the cost of creating and committing transactions over those transports. Such cost is relevant because it appears in every transaction started by some remote client and successfully completed.

---

[3]The OTS context contains not only a CORBA reference to the transaction `Coordinator`, but also a second reference, which identifies the transaction `Terminator`.

Since in our experiment the transactions were created and immediately committed, they did not involve any transactional resources. As a result, the two-phase commit protocol did not run, and transaction completion was a very fast process.

Experimental results. Fig. 4a shows the average time spent creating and committing a transaction over a selected transport. Each bar in that figure has two parts: the bottom part corresponds to the time employed to create a new transaction; the upper part corresponds to the time employed to commit the transaction. The whole bar therefore represents the overall time spent in transaction demarcation. Fig. 4b presents the detailed results.

As Fig. 4a shows, JBoss Remoting again presented better performance than IIOP and SOAP/HTTP. At the root of this better performance is JBoss Serialization, the very efficient marshalling/unmarshalling mechanism used by JBoss Re-
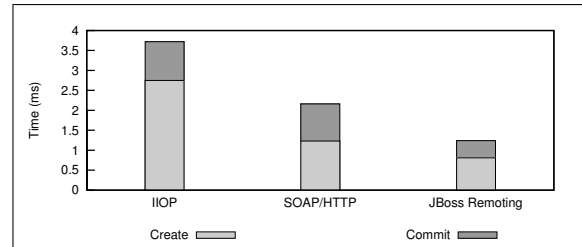
(a) Average demarcation times.

		Create	Commit	Total
IIOP	Average time	2.75 ms	0.97 ms	3.72 ms
	Data transfer	1,168 bytes	89 bytes	1,257 bytes
SOAP/HTTP	Average time	1.23 ms	0.93 ms	2.16 ms
	Data transfer	1,758 bytes	1,000 bytes	2,758 bytes
JBoss Remoting	Average time	0.81 ms	0.43 ms	1.23 ms
	Data transfer	1,661 bytes	682 bytes	2,343 bytes

(b) Detailed results.

Figure 4. Results of the third experiment.

moting. Surprisingly, Fig. 4a also shows that SOAP/HTTP was faster than IIOP in this experiment. The reason, again, is the marshalling and unmarshalling of CORBA references. Whenever a client creates a new transaction, a transactional context is also created and is returned to the client. As we have already seen in 4.3, the transactional context defined by CORBA OTS contains two CORBA references, and the marshalling and unmarshalling of those references is a slow process.

The presence of CORBA references in the transactional context is what makes the creation of a transaction over IIOP much slower than the creation of a transaction over SOAP/HTTP (2.75 versus 1.23 ms). On the other hand, commit messages carry no transactional context. If we consider the transaction commit operation, we see that it consumes about the same time over IIOP and over SOAP/HTTP (0.97 and 0.93 ms, respectively).

4.5. Two-phase commit overhead

The experiments presented so far did not involve any transactional resource. In order to evaluate the total overhead of distributed transactions, we performed an experiment that actually runs the 2PC protocol.

Our fourth experiment employed a simple distributed application, which comprises an EJB client and two identical EJB components[4]. We ran the EJB client as a stand-alone application, on its own machine, and deployed each of the EJB components on a distinct JBossAS instance. Each of these server instances ran on a separate computer. A fourth machine hosted another JBossAS instance, whose transaction manager (an instance of XActor) played the role of transaction coordinator. Each of the EJB components kept persistent data on a separate DBMS, as shown in Fig. 5. We used the Derby DBMS in its embedded mode and employed Derby's "embedded JDBC driver", so that each of the DBMS instances ran in the same virtual machine as the JBossAS instance that contains the corresponding EJB component.

The EJB client performs the following steps (Fig. 5):

1. it creates a new transaction,
2. it invokes a method of the EJB on the first server, triggering a write operation to a database in the first DBMS,
3. it invokes a method of the EJB on the second server, triggering a write operation to a database in the second DBMS, and then
4. it commits the transaction.

Steps 2 and 3 involve the propagation of a transactional context along with the remote invocations targeted at the EJBs.
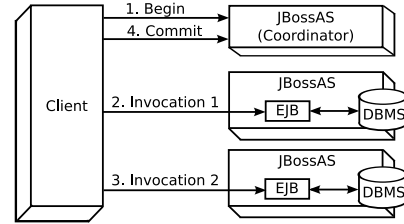

Figure 5. The fourth experiment.

In those same steps, each database write triggers the registration of an application server[5] with the transaction coordinator. Finally, in step 4, the coordinator runs the 2PC protocol.

We performed this experiment using IIOP, SOAP/HTTP, and JBoss Remoting. For each transport, we have actually performed three variations of the experiment and measured the duration of the steps above in the following scenarios:

 (i) using no transactions at all (steps 1 and 4 were skipped in this scenario),

 (ii) using transactions, but with the transaction log disabled, and

(iii) using transactions, with the transaction log enabled.

In the first scenario, steps 2 and 3 do not propagate a transactional context (since no transaction was created) and thus no application server registers remote `Resources` with the coordinating server instance. Moreover, scenario *(i)* does not include a 2PC run. Since scenario *(i)* comprises only a pair of non-transactional invocations to EJB methods, it provides a yardstick against which the transactional scenarios *(ii)* and *(iii)* can be compared. The two transactional scenarios allow us to evaluate the impact of the transaction log on the overall cost of distributed transactions.

Experimental results. For every transport, Fig. 6a presents a set of three bars. The leftmost bar of each set (in white) corresponds to scenario *(i)*. It represents the total time spent on steps 2 and 3, with no transactions involved. (Recall that steps 1 and 4 are not executed in scenario *(i)*.) The bar in the middle corresponds to scenario *(ii)*. It shows the time spent on steps 1 to 4 when transactions are employed, but the transaction log is disabled. Finally, the rightmost bar of each set corresponds to scenario *(iii)*. It reports the time spent on steps 1 to 4 when transactions are employed and the transaction log is enabled. Fig. 6b shows the detailed results for scenario *(iii)* (transactions employed and transaction log enabled).

As Fig. 6a shows, JBoss Remoting was the fastest transport in our three experimental scenarios. This result makes sense, as JBoss Remoting was the fastest transport for trans-

---

[4]Those EJB components are different from the one used on the previous experiments.

[5]More precisely, each database write triggers the enlistment of an application server's transaction manager (an instance of XActor) as a remote `Resource` registered with the transaction coordinator (also an instance of XActor).
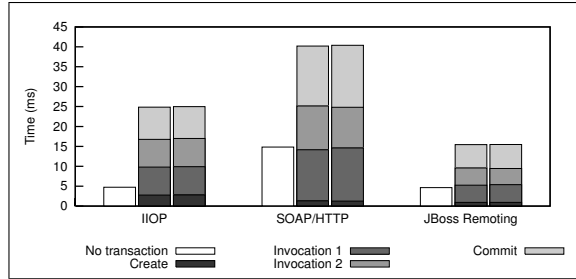
(a) Average execution times

	Create	Invocation 1	Invocation 2	Commit	Total
IIOP	2.80 ms	7.11 ms	7.07 ms	7.99 ms	24.97 ms
SOAP/HTTP	1.27 ms	13.34 ms	10.22 ms	15.57 ms	40.40 ms
JBoss Remoting	0.94 ms	4.43 ms	4.08 ms	6.03 ms	15.49 ms

(b) Detailed results for scenario *(iii)*.

Figure 6. Results of the fourth experiment.

actional remote invocations (Fig. 3a) and for transaction demarcation (Fig. 4a). Therefore, it is not surprising to see JBoss Remoting also as the fastest transport when the 2PC protocol actually runs. IIOP presented a considerably slower performance when compared to JBoss Remoting, but it was still noticeably faster than SOAP/HTTP.

We can also see in Fig. 6a the cost of transaction creation was largest in the IIOP case. Again, this fact was due to marshalling/unmarshalling of the CORBA references contained in the OTS context returned by the operation that creates a new transaction. For the same reason, the EJB invocations over IIOP were considerably slower than their counterparts over JBoss Remoting: all those invocations carried the transactional context, but in the IIOP case the propagated context contained CORBA references. SOAP/HTTP once again presented the worst execution times, due to its use of XML-based communication.

Fig. 6a also shows that the use of the transaction log did not impose a significant overhead on the execution of a distributed transaction. The difference between the second and third bars of each transport was minimal. This means that the cost of force-writing transaction log records to disk is much smaller than the communication costs (marshalling, unmarshalling, and context propagation included) of all the interactions between the parties involved in a distributed transaction, regardless of the transport employed. In the end, the communication overheads dominate the execution time and the 2PC logging activity becomes irrelevant.

Finally, Fig. 6a shows that the use of distributed transactions imposes a very high overhead, independently of the selected transport. If we look at the IIOP set of bars, we see an execution time of 4.74 ms when no transactions were employed. Since this time increases to 24.97 ms when we employ transactions and enable the 2PC log, the transactional overhead is 427%. In the case of SOAP/HTTP, the execution time increases from 14.84 ms to 40.40 ms: an overhead of 172%. For JBoss Remoting, the transactional overhead is 233% (a time increase from 4.65 ms to 15.48 ms).

5. Related work

Very little has been published about the performance of transactional remote invocations. There is, however, plenty of research on the performance of plain (non-transactional) remote invocations. In [7], Elfwing *et al.* compare the performance of remote invocations over SOAP/HTTP and IIOP. The authors identify some reasons for the poor performance of SOAP/HTTP and suggest ways of improving SOAP/HTTP implementations. In [5], Davis and Parashar evaluate the performance of five SOAP stacks and identify sources of inefficiency in them. The distinguishing points of their work are: *(i)* it evaluates several SOAP stacks and *(ii)* it compares the performance of non-transactional remote invocations via SOAP/HTTP, CORBA, and Java RMI. In [6], Demarey *et al.* present a benchmark to evaluate the round-trip times of non-transactional remote invocations. They apply that benchmark to a set of middleware platforms that includes five IIOP-based ORBs, one SOAP/HTTP stack, and two Java EE servers.

Previous research on the performance of transactional remote invocations is scarce and focused exclusively on the IIOP transport. In [9], Gorton *et al.* examine three implementations of the CORBA OTS specification and present a valuable comparison between them. The authors also assess the cost of transaction demarcation over IIOP. The two major differences between their work and ours are: *(i)* we examine other transactional costs besides demarcation, and *(ii)* we evaluate those costs not only for the IIOP transport, but also for SOAP/HTTP and JBoss Remoting.

In [20], Tran and Gorton discuss the scalability of transactional CORBA applications. By using a simple transactional application, the authors evaluate the impact of simultaneous clients on the performance of a particular OTS implementation.

6. Concluding remarks

This paper considered transactional remote invocations in single-client scenarios. Our ongoing work addresses scalability issues. We are currently investigating the behavior of the same transport protocols (IIOP, SOAP/HTTP and JBoss Remoting) when transactional invocations are concurrently performed by a growing number of remote clients.

The performance cost of transactional remote invocations has three components: transaction demarcation, propagation of the transactional context along with each invocation, and two-phase commit. Most of the first component can be avoided if transaction demarcation takes place within an application server. This indeed happens in many Java EE scenarios, either with the usage of container-managed transactions or with the explicit demarcation of transaction boundaries by application components deployed in the Java EE server. The other two components of the

transactional cost are unavoidable. Note that the third component comprises both the 2PC execution, at commit time, and the dynamic registration of transaction participants with the 2PC coordinator.

Our last experiment allowed us to evaluate the total cost of transactional remote invocations in a very simple distributed application. It is interesting to look at the breakdown of that cost into its three components. The transactional overhead of 427%, observed in the IIOP case, is the sum of the following three components: a demarcation overhead of 80%, a context propagation overhead of 65%, and a 2PC overhead of 283%. For SOAP/HTTP, the overall cost of 172% corresponds to a demarcation overhead of 15%, plus a context propagation overhead of 13%, plus a 2PC overhead of 145%. For JBoss Remoting, the transactional cost of 233% corresponds to a demarcation overhead of 29%, plus a context propagation overhead of 16%, plus a 2PC overhead of 187%.

We believe that the high overheads in the IIOP case could be substantially reduced by optimizations on the ORB code that marshals/unmarshals CORBA references. Reference marshalling/unmarshalling takes a heavy toll not only in transaction demarcation and context propagation. It also accounts for significant part of the 2PC overhead, as the OTS operation that registers a transaction participant with the 2PC coordinator receives a CORBA reference as input parameter and returns another reference to its caller.

In the SOAP/HTTP case, the relatively low overheads are probably due to the highly optimized implementation of the WS-C and WS-AT specifications by XActor's SOAP/HTTP plug-in [18]. Even so, the SOAP/HTTP transport still has the worst execution times. XML parsing is at the root of the bad performance of that transport.

The usage of the Derby DBMS in "embedded mode" certainly affected the relative overheads that we observed. Running the DBMS in the same process as the application server makes the database accesses faster than what would be possible with a separate database server. If we had placed the DBMSs on separate machines, the EJB invocations would take longer and the cost of transactional remote invocations would be smaller in relative terms. Nevertheless, that cost would still be very high. Transactional remote invocations should therefore be used wisely. They are indispensable in many situations, but their overuse can severely impair the performance of a distributed application.

References

[1] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Differential serialization for optimized SOAP performance. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pages 55–64. IEEE Computer Society, 2004.

[2] Apache Derby website, 2008. `http://db.apache.org/derby/`.

[3] G. Brose. JacORB: Implementation and design of a Java-ORB. In *Proceedings of DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems*, pages 143–154, 1997.

[4] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, pages 246–254. IEEE Computer Society, 2002.

[5] D. Davis and M. P. Parashar. Latency performance of SOAP implementations. In *Proc. of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CC-GRID '02)*, pages 407–412. IEEE Computer Society, 2002.

[6] C. Demarey, G. Harbonnier, R. Rouvoy, and P. Merle. Benchmarking the round-trip latency of various Java-based middleware platforms. *Studia Informatica Universalis*, 4(4):7–24, May 2005.

[7] R. Elfwing, U. Paulsson, and L. Lundberg. Performance of SOAP in web service environment compared to CORBA. In *Proc. of the 9th Asia-Pacific Software Engineering Conference (APSEC '02)*, pages 84–96. IEEE Comp. Soc., 2002.

[8] M. Fleury and F. Reverbel. The JBoss extensible server. In *Middleware 2003 — ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *LNCS*, pages 344–373. Springer-Verlag, 2003.

[9] I. Gorton, A. Liu, and P. Tran. The devil is in the detail: A comparison of CORBA object transaction services. In *Proc. of the 6th Intl. Conf. on Object Oriented Information Systems (OOIS '00)*, pages 211–221. Springer-Verlag, 2000.

[10] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for an evaluation of RMI protocols for scientific computing. In *Proc. of the 2000 ACM/IEEE Conf. on Supercomputing*. IEEE Comp. Soc., 2000.

[11] JBoss Remoting website, 2008. `http://labs.jboss.com/portal/jbossremoting/`.

[12] JBoss Serialization website, 2008. `http://labs.jboss.com/serialization/`.

[13] JBoss Web Services website, 2008. `http://labs.jboss.com/jbossws/`.

[14] OASIS. *Web Services Atomic Transaction 1.1*, April 2007.

[15] OASIS. *Web Services Coordination 1.1*, April 2007.

[16] Object Management Group. *CORBA Transaction Service Specification, version 1.4*, March 2003.

[17] F. Reverbel and I. Silva Neto. Dynamic support to transactional remote invocations over multiple transports. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC 2008)*, pages 499–506. ACM Press, 2008.

[18] I. Silva Neto and F. Reverbel. Lessons learned from implementing WS-Coordination and WS-AtomicTransaction. To appear in *Proc. 7th IEEE/ACIS Intl. Conf. on Computer and Information Science (ICIS 2008)*, Portland, USA, 2008.

[19] Sun Microsystems. *Enterprise JavaBeans, Version 3.0: EJB Core Contracts and Requirements*, May 2006.

[20] P. Tran and I. Gorton. Analyzing the scalability of transactional CORBA applications. In *Proc. of the 38th Intl. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS '01)*, pages 102–110. IEEE Comp. Soc., 2001.

[21] Woodstox website, 2008. `http://woodstox.codehaus.org/`.