An Evaluation of the Middleware’s Impact on the Performance of Object Oriented Distributed Systems

Christos Tselikis\textsuperscript{a} Sarandis Mitropoulos\textsuperscript{a}\textsuperscript{*} Christos Douligeris\textsuperscript{a}

\textsuperscript{a}Department of Informatics, University of Piraeus, 80 Karaoli and Dimitriou Str., Piraeus 185 34, Greece

Elsevier use only: Received date here; revised date here; accepted date here

Abstract

In this paper we present a performance analysis of the response time of a client server e-banking application running over three different enterprise middleware platforms, namely HTTPServlets, RMI and Web services with JAX-RPC. We conducted performance testing with the purpose to reveal the specific characteristics of each middleware technology and the impact that they infer on the distributed application’s performance. A server node running the three J2EE platforms was benchmarked over a wide array of intranet usage patterns. A statistical analysis of the collected data led into conclusions regarding the benefits of each middleware technology. The simulation framework can be further extended to become a testing tool able to differentiate on various service demand classes as an input in distributed applications so as to offer first-cut validated results concerning the systems’ performance.

© 2006 Elsevier Science. All rights reserved

Keywords: client-server interactions; middleware software; usage patterns; emulation.

1. Introduction

The design of the appropriate distributed models for facilitating the communications in today's Internet has attracted considerable attention, both in the research and the commercial world.

E-commerce infrastructures and globally distributed interactive content provide the end-user with the means to easily access useful information and the businesses with the facilities to trade and procure electronically.

As reported by the European e-Business Market Watch (2005), numerous cross-sector and cross-country surveys indicate that the usage patterns in business environments play a significant role in the adoption of new systems and services.

Also, the presence of advanced communication infrastructures and service features leads to increased transaction volumes and application variety.

In such diverse environments there is a clear necessity to develop large-scale internet and intranet applications in the most efficient way in terms of scalability, responsiveness, interoperability, reusability, maintainability and economics.

To our view, the key factor in achieving the above objectives is the underlying middleware infrastructures, most often based on object technologies, which provide for an open interface to the network services of the Internet.

As new Web frameworks appear and evolve, like the various Java2 Enterprise Edition (J2EE) compliant application servers, the Microsoft’s .Net

\begin{itemize}
  \item Corresponding author. \textit{E-mail address: }sarandis@unipi.gr
  \item This work was funded by the PENED Research Program
\end{itemize}
platform, or the numerous implementations of the CORBA standard, it is evident that performance testing should keep in pace with the challenges that these new technologies set. It is also obvious that the modern needs in performance evaluation should be addressed in a reliable way.

In this paper we focus our attention on J2EE components aiming to study the performance and scalability potential of various middleware platforms. First, we experiment with live systems and then we consider these platforms as open systems and not, as it is usually the case, as black boxes. Finally, we investigate the role of the underlying communication models.

To this end, a testing framework has been developed that is able to test object oriented distributed technologies across heterogeneous platforms, by applying various types of loads on them. Ultimately, this tool will be able to handle various actors possessing multiple roles during their interaction with the distributed subsystems.

We believe that this approach will be exploitable and beneficial for new emerging technologies, such as grid-based applications or asynchronous Web frameworks.

This paper is organized in sections as follows. Section 2 cites and comments related work. In Section 3 we present three generic synchronous distributed models in the Web. Section 4 describes the test application’s design. Section 5 states the analytical model and its limitations in describing the distributed system at hand. In Section 6 we define four likely usage patterns of the software. In Section 7 the emulation environment is described in Unified Modelling Language (UML) notation. Section 8 presents specific overhead factors in the middleware stacks. Section 9 presents the experimental results and an extended interpretation of them, while Section 10 supplies hints for future work.

2. Related Work

In a five-year forecast presented by Pancake (1995) it was predicted that the object-oriented technologies would attract the future interest of industries in developing large scale distributed systems. An obstacle spotted at that time was the lack of widely adopted frameworks and tools that could render the object technologies able to support performance-sensitive applications. Today, the need to integrate performance modelling in the development of software-based industry applications still remains.

Previous related work conducted by Franklin et al. (1997) used client DBMS workloads and an algorithms’ taxonomy for cache consistency in client-server transactions, while Gelenbe (1999), Wu (1999) and Haring et al. (2000) have addressed various performance analysis issues.

In the comprehensive methodology presented by Menascé and Gomaa (2000) performance modelling parameters were integrated with the design of a database-centric client-server system on the perspective of estimating the usage of hardware-related resources such as disk, network and CPU times in statements execution. The consumption of software-related resources was bound either by the transaction specifications defined in the specific system examined or by DBMS-related functionalities (query optimization and access plans).

There is a need, though, to contemplate on the performance impact of the middleware layers for open communications between remote sites. Furthermore, there is a need to enhance the set of workloads and service demand classes that compete for the distributed systems’ resources. This is addressed in this paper by introducing the asymmetry workload factor $f$.

Bansal and Dalton (2001) benchmarked and profiled light transactions with Web services on wireless handheld devices. In contrast to our scenarios the bandwidth-constrained 802.11b wireless link was the critical bottleneck in these experiments while the small devices’ processing power had a lesser impact. It was observed that web services with direct SOAP messaging performed very slowly (the request completion time took 11 seconds). The client side proxy architecture was chosen, because it was found to be faster. To alleviate the overhead in the limited resources environment of small devices, we also propose the optimization of the SOAP protocol as well as client implementations in C, instead of Java. Bansal and Dalton (2001) found that the C client version was thirty times faster than the Java one. The above are indications that the role of language, design patterns and application-level architectures needs to be further analyzed.

The performance comparison conducted by Garcia-Sanchez, F., et al. (2005) concerned the sequential vs. the multi-threaded model of execution in implementations based on the stream services offered by the ACE / TAO CORBA A/V framework. These tests were performed on Linux and verified the increased delay and especially the jitter caused by the multi-threading model in execution of real-time applications.

Viloldo and Serrat-Fernandez (2005) investigated the possibility of using plugins coded by clients for service adaptation. They compared the performance
of a base Web services stack with one that integrates an adaptation layer for binary Simple Object Access Protocol (SOAP) encoding, while varying the level of concurrency and the response message size. Although the level of concurrency was kept low it was proven again that the round trip delay in the case of plain SOAP is considerably larger than that with the binary plug-in.

Chen et al. (2005) tackled the intense need for predicting the behaviour of component-based distributed systems before their large-scale deployment. They devised a performance cost model that can predict metrics, such as the response times, that clients experience when they contest for simultaneous database accesses. Performance was examined in regard to the number of parallel threads being present in the client workload and the container’s thread pool.

The need for a detailed performance parameter definition, in order to efficiently test the unique behaviour of each component-based middleware technology, was stressed in this work.

3. Synchronous Distributed Models in the Web

A significant part of Web-based distributed infrastructures implements client-server interactions over the request-reply protocol. Modern distributed applications and systems can be distinctly differentiated by whether they follow the synchronous or the asynchronous communications model for message delivery and request execution.

A comparative performance study analyzes the asynchronous Message Oriented Middleware model versus implementations of the synchronous communications model with Remote Procedure Calls (RPC) (Menasce, 2005).

In this paper, we focus our attention on the client-server paradigm carried out by Java components that communicate synchronously in the three-tier architecture. We made this choice as a first step in developing a testing suite able to accept various technologies as input and, secondly, because we are interested in performance issues of time-constrained applications, as is our on-line test application.

The three tiers for information retrieval in the example banking system that we use as a model for this work are:

- The database layer where we hold the personal account information.
- The business logic layer, i.e., the available middleware platforms, message managers, transaction processing monitors and application servers.
- The user applications layer, i.e., the clients that submit queries about personal account information, or about product and partner information.

We picked the following synchronous middle-tier models as common candidates in such an environment with the objective to underline the distinct approaches and concepts followed in each one of them:

- **The stream-oriented communications model.** Processes perform input and output over bi-directional and synchronized byte-stream connections. Examples of this generic model include the information exchange over raw TCP sockets and HTTP sessions between browsers and server-side Servlets in the Web tier.
- **The distributed object model.** We consider Remote Method Invocation (RMI) as an adequate example of the distributed object model. Information is passed as arguments and return values between objects that are located on different machines.
- **The Services Oriented Architecture.** For the “publish, discover and invoke” model in the Web services tier, we chose the XML-based Java Application Program Interface for XML-RPC (JAX-RPC) to synchronously transfer transactional data over SOAP.

A diversity of commercially available middleware packages (open-source or proprietary) supports the above models. Each one of them possesses individual characteristics. We tested three specific J2EE packages, the relevant Application Program Interfaces of which will be presented in more detail in Section 8. However, it is worth drawing some differences in advance.

Both RMI and JAX-RPC are implementations of service access through remote interfaces. Another essential distinction is that Servlets and Web Services require an application server’s environment for HTTP/SOAP message handling and request execution, while RMI can serve requests solely based on the Java run time system services for remote method call and execution. Furthermore, RMI, as an object oriented RPC technology, is closer to the object-distributed model,
though it only supports an elementary name service achieved by distributing the RMI registries.

On the other hand, Servlets and Web services assume the existence of a central manager process that encapsulates, protects and allocates shared resources to clients upon invocation, an approach that resembles traditional distributed systems principles.

At the language level, Web services and JAX-RPC implementations differ from the distributed object model in that they (should) lack main methods and servant constructors. To our view this is strongly related to their execution inside the application server engines.

Our intention is to investigate how synchronization is achieved by different middleware platforms and to highlight the middleware effect on the responsiveness of synchronized applications in the Web.

Furthermore, via the use of simulation experiments our objective is to build appropriate usage profiles and empirically test inherent middleware features that are supported in each case.

4. Test Application’s Design Patterns

The test application is a distributed client-server banking application with on-line transactions processing characteristics. On-line transactions over the Internet comprise of frequent, short-duration (in the order of up to thirty seconds across the Internet) and small message exchange in rapid database updates. The validation and update of a credit card is a characteristic example of such a transaction.

Transactions, for performance reasons, are preferably synchronised in a session and most often follow the synchronous RPC communications model, i.e., the client process blocks until it receives the server’s result. On the other side, the server process might also stay synchronized to subsequent client requests, in case that the transaction specifies more than one interaction. The choice of a synchronized e-banking test application is appropriate in testing synchronous middleware platforms.

The banking transaction is designed to be composite, i.e., it consists of two atomic operations. The first operation is the authentication of the user. The authentication is carried out in the server by searching the user_id in the bank database. If this information is found, the user is presented with the current balance of his account. Following the successful completion of the authentication operation, the second atomic operation performs either a debit or a credit on the client’s balance, according to the “action” parameter that he submits in his second request to the system. Accordingly, the server updates the balance in memory, writes the updated value to disk and, finally, informs the user of the new status of the account.

The sequence of the query-response message exchange is shown in Fig. 1.

```
1. client sends his PIN number to server
2. server authenticates the user:
3. if user_id is valid
4. then server reads balance from memory,
5. server sends to user balance and date
6. else server sends an error message to client
7. client sends a request for a debit or credit to server
8. server checks the validity of typed amount
9. if amount is valid
10. then checks if it is a debit or credit request
11. if debit
12. then server reads user’s balance from memory,
13. if credit limit is sufficient
14. then modifies balance in memory
15. writes new balance to disk file
16. sends new balance to client
17. else notifies the user to try again
18. if credit
19. then server reads user’s balance from memory
20. modifies balance in memory
21. writes back new balance to disk file
22. sends new balance and current time to client
23. else sends prompt message to user to try again
```

Fig. 1 The composite transaction specification.

In building the test application we adopted certain software design patterns that we kept common in all...
• The **user reaction time**. It is kept minimal because we simulate machine-to-machine communications, so we don’t want to let the response times be dominated by a probabilistic model of the users’ silent times.

• The **multi-threaded** server. Jointly with File System permissions, the concurrency control for consistency and integrity is secured in the software by synchronizing access to shared memory objects with preemptive scheduling. Hence, no two clients are allowed to access data items at the same time.

• The **static proxy** design pattern. This is a mediator component at the client side that offloads the end user application by forwarding messages to the remote endpoint, according to the underlying middleware protocol rules.

• The **object serialization** mechanism that is used in Servlets, RMI and JAX-RPC to pass remote object references on the wire. Relational data is encapsulated in packets of the JavaBeans **ValueType** structure. Object serialization facilitates the exchange of structured and large-size RPC messages sent over TCP. The latter implements the necessary flow control mechanism to accommodate message reception.

• The **disk access** pattern. It is common in all our scenarios. The server keeps relational data (user_id and balance information) managed by synchronizing multiple accesses to them. Also, the serving process, as being a client of the Operating System services, is designed to access the disk only once and at start-up to load all the necessary data objects into memory. This type of RAM caching is implemented with a Hash Table container. In effect, upon client requests, data reading is performed with the minimum possible number of disk accesses, for disk accesses are the critical performance factor in data-centric systems. On the other hand, the test application is write-intensive since, upon destruction of each thread, the updated balance is written to the disk and also is backed up in different memory, which is designed to act as a hot stand-by.

• The **disk-less** client machines that perform minimal processing with no caching.

### 5. Analytical Model

Our distributed system consists of multiple clients that form a **Poissonian** stream of WWW requests with arrival rate $\lambda_s$. The incoming requests share a single communication link and the CPU server are assumed to respond with exponential times of mean service rate $\mu_s$.

We focus on the test application’s response time because we consider it to be the most indicative performance metric in terms of the overhead the middleware imposes.

The distributed application’s response time $T_{RT}$ is the elapsed time from the submission of a client transaction request until the time when the server reply is received back by the client. This includes a negligible processing latency at the client side, the round-trip time $T_N$ spent over the network plus a significant processing time $T_S$ spent inside the remote service endpoint. Equation (1) describes the above in the case of the composite transaction of two remote operations:

$$T_{RT} = 2 \times (T_N + T_S) \tag{1}$$

$T_S$ is the request’s mean sojourn time, i.e., the time spent inside the server waiting for service plus the service time that the allocated hardware and software resources actually need to commit the operation. Hence, $T_S$ can be written as in equation (2):

$$T_S = T_{CPU+NIC+DiskIO} + T_{Middleware} \tag{2}$$

We attribute the hardware overhead to resources such as the server’s processor power, disk and network cards. $T_{Middleware}$ in Equation (2) represents the mean latency imposed by software components, such as the client and server middleware elements that handle the incoming and outgoing messages.

Assuming that the server’s buffer memory is unlimited and that the arrival rate $\lambda_s$ is independent of the number of requests already present in the system, we can accept the $M/M/1$ open queuing model as a basic analytical model. This predicts that, as long as $\lambda_s < \mu_s$, the mean sojourn time is:

$$T_S = \frac{1}{\mu_s - \lambda_s} \tag{3}$$
Equation (3) can be re-written in the following way:

$$T_S = T_{S1} + T_Q$$

(4)

where,

$$T_{S1} = \frac{1}{\mu_S}$$

is the mean service time that a single transaction needs to commit when no other customer request is present in the system and latency

$$T_Q = \frac{\rho}{\mu_S - \lambda_S}$$

(5)

is the mean “waiting time” experienced inside the multi-threaded server given that the system utilization factor $\rho = \lambda_s / \mu_s$ is smaller than unity. $T_Q$ in these terms analytically describes the multi-threading model for service call and execution. Equation (5) predicts that the mean “waiting time” per client request varies asymptotically with respect to the requests arrival rate $\lambda_s$, or, equivalently, to the system resources utilization factor $\rho$.

Furthermore, by application of Little’s law, we derive that, in the steady state, the mean delay depends on the total number of the customers present in the system ($N$) and their arrival rate ($\lambda_s$). For this reason, these two parameters were the first two of the workload parameters that were chosen to be used in the simulation experiments.

It is apparent though, that Equation (5) cannot adequately model the dynamic behaviour of software components that interact with each other. Besides, it can not explicitly model the dependency of the requests’ mean “waiting time” on the message length that is exchanged between the communicating parts, since the mean service rate $\mu_s$ in equation (5) is assumed fixed (Dunlop and Smith, 1994).

The simulation measurements will reveal the effect that different middleware platforms incur on the test application’s response time and the server’s throughput. Furthermore, the experiments will show the load’s concurrency level for which each middleware technology yields its best responsiveness.

6. Workload Parameter Setting

As the Web scales we have to classify the usage patterns that we encounter in intranets and internets. First, we categorize the sources of Web traffic on the number of users connected in enterprise workgroups.

Table 1 shows the Small Office Home Office (SOHO) usage profile that we considered. This profile has no more than ten connected workgroup members.

The next usage is that of a SE (Small Enterprise) of up to 50 workgroup members. The Small Medium Enterprise (SME) usage profile usually corresponds to traffic generated by no more than 250 employees being connected through a C-class network. We examined two cases as for the SME usage profile, namely one with 100 employees (SME1) and another one with 150 clients connected (SME2).

Consistently with the discussion done in the previous Section, as shown in Table 1, the above workgroups are characterized by their requests arrival rate $\lambda_s$.

The size of messages has already been identified as a factor that affects the server’s throughput or service rate $\mu_s$. Medium size messages utilize efficiently the available capacity of the CPU and the Ethernet bandwidth. Heavier messages saturate the resources to a point where no more requests can be served leading to successive transaction rejections.

As shown in Table 1, we controlled the payload size as being the third simulation parameter $L$ in the range between 75kBytes and 150kBytes.

Apart from these criteria for workload characterization we also incorporated in our experiments design the asymmetry factor $f$, an index of the asymmetry between the data volume transferred at the upload link and that downloaded on the reverse link during a client-server session.

It is well known that on the upload path a single Internet user most often sends interrogating traffic in the form of URL requests. This data is remarkably less than the amount of data that he usually downloads. The asymmetry depends on the content (audio, images, video) the requested page contains. On the contrary, enterprise applications are often symmetric in the data volumes exchanged and the bandwidth consumed in the two communication paths (two indicative examples are business to business transactions, even VoIP applications).

The use of $f$ captures a characteristic that varies across typical applications in today’s Internet. By measuring the application’s responsiveness for values of $f$ in the range $[0, 1]$ we cover many possible combinations of the message sizes exchanged between client and server.

Table 1 shows the values and ranges for the parameter set ($\lambda_s$, $N$, $L$, $f$) that we used in the test plan.
Table 1 Workload parameter values and ranges

<table>
<thead>
<tr>
<th>Workload parameter</th>
<th>SOHO</th>
<th>SE</th>
<th>SME1</th>
<th>SME2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency level N</td>
<td>10</td>
<td>50</td>
<td>100</td>
<td>150</td>
</tr>
<tr>
<td>Arrival rate λ_k</td>
<td>2.0</td>
<td>2.25</td>
<td>2.50</td>
<td>2.75</td>
</tr>
<tr>
<td>Message length L</td>
<td>75 kB-750 kB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asymmetry factor f</td>
<td>0.1-0.2</td>
<td>0.3-0.5</td>
<td>0.6-0.8</td>
<td>0.9-1</td>
</tr>
</tbody>
</table>

7. Emulation Environment

Pooley and King (1999) investigated the potential use of the Unified Modelling Language (UML) sequence and state diagrams in software performance estimation. In particular, they considered iterations in sequential behavior and they used simulation traces. They reported that reasonable ordering of events breaks down at large parallel loads. Furthermore, the direct simulation of UML collaboration and activity diagrams were suggested as a solution to predict performance, as they both can provide valid system views. Appropriate libraries were created to achieve this purpose.

In our study we are interested in experimenting with live distributed systems (Roper, 2001). Our aim is to build a testing framework (with volume testing points) which can accept as input simulation parameters which characterize various loading conditions, apply them on prototypes and consequently offer valid predictions of the real systems performance before these systems are developed in large scale.

To achieve the above goals, we extended the JavaSim Object Oriented and Discrete Event simulation package (Little, 1998) with transactions capabilities. Following the philosophy of JavaSim we considered the parallel requests to come from client entities which extend the JavaSim base simulation classes and which contact the remote server entity as processes that run in parallel (process-based simulation).

We created three J2EE versions of a client-server application running in an emulation environment that consists of two separate machines. The interactions between the distributed components adhere to the test application’s specification.

Our simulation application is controlled by the Transactions Source module that starts simulation, instantiates the Poissonian stream which loads the banking server, lets the parallel clients interact with the server and, then, calculates the mean of all the response times, before terminating the simulation. In UML notation the Sequence Diagram of Fig. 3 illustrates the SME load and the sequence of interactions included in one successful (committed) transaction. The diagram of Fig. 3 corresponds to the RMI scenario.

As shown, the SME load is considered to be a client-side actor in the business model, an instance of the generic class Arrivals. This actor interacts with the system and iteratively creates 100 RMIClient objects, each one identified by the user_id parameter and each one sending the parameter “action” to designate the desired remote banking operation to be performed.

The server is considered to be a UML element of type <<subsystem>>. It implements the authentication and debit or credit operations according to their signatures declared in the remote interface BankingIF.

It is obvious that it is impossible to depict in a UML diagram the knitted timing sequence of all the concurrent events that take place. Instead, Fig. 3 only shows a snapshot of the authentication (650 msecs) and response times (1.375 msecs), as these samples are recorded in a simulation trace for the SME workload case.

The Transactions Source also generates streams of HTTP and SOAP messages for the Servlets and JAX-RPC implementations respectively. Although the sequence of interactions is identical, the service discovery, invocation and internal processing steps depend on the particular middleware layers and the architecture supported inside the bank server in each case.
8. The Middleware as Software Overhead

We used two separate but identical in capabilities machines on which we ran the three versions of the transactions source (client side) and the three versions of the e-banking server (server side). The hardware and software details of the configuration are shown in Table 2.

### Table 2

<table>
<thead>
<tr>
<th>Middleware</th>
<th>Servlets</th>
<th>RMI</th>
<th>JAX-RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>APIs and models</td>
<td><a href="jakarta.apache.org/tomcat-docs/v7.0/api/jsp-api.html">Servlet api</a> as included in the Jakarta Apache Tomcat 1.4 Servlet / JSP container.</td>
<td><a href="http://java.sun.com/products/jdk/1.2/docs/guide/jini/rmi/api/rmi-api.html">RMI API</a></td>
<td><a href="http://java.sun.com/products/jdk/1.5/docs/guide/xml/web-tier-api.html">JAX-RPC api</a> which is included in the Sun's XML-based JWSDP toolkit. In support of the SOAP 1.1 specification, we integrated JWSDP with Tomcat.</td>
</tr>
<tr>
<td>Container</td>
<td>Tomcat</td>
<td>JVM</td>
<td>Tomcat integrated with Sun's JWSDP SOAP kit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client/Server</th>
<th>OS</th>
<th>hardware</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows XP</td>
<td>Professional 7 OS</td>
<td>1 GHz Pentium III, 1GB RAM, 15GB Virtual Memory</td>
<td>100 Mbps Fast Ethernet</td>
</tr>
</tbody>
</table>

In this testbed we deployed the three J2EE packages as extensions to the core J2SDK APIs. The API libraries shown in Table 2 are:
In the followings we attribute the middleware overhead to a list of mechanisms related to information handling, which jointly with the underlying operating system services affect the test application’s performance. In a top to down listing, the most significant mechanisms are:

- **The message handling mechanism.**
  At the front of the application server message handlers constitute the critical interface between the Internet incoming traffic and the middleware engines further inside the server. In this sense, message handlers, for example SOAP actors, initiate system-level policies (e.g. authentication), upon reception of a SOAP message. We offer some specific characteristics of each framework bearing in mind that they are vendor-dependent:
  - **Servlets.** The handling of HTTP packets is performed inside the Coyote Connector. HTTP parameters of a GET or POST packet reaching the Coyote component of the Application server are then passed inside the servlet engine and finally to the BankingHttpServlet object for appropriate processing.
  - **RMI.** In Java RMI message marshalling and un-marshalling are performed by stubs and skeletons created by the RMI compiler. RMI arguments and return values are transformed to a format suitable for transmission over the wire.
  - **JAX-RPC.** The front-end component that accepts SOAP messages is the JAXRPCServlet component. This element strips XML out of the body of a SOAP message. Then the XML information is dispatched to the appropriate pair of Tie and Java Implementation objects. The necessary parser API, used by default in Sun’s JWSDP, is the Java API for XML Processing (JAXP).

- **The multi-threading and scheduling models**
  which are supported at the application server, the middleware run time system and, finally, at the lower socket level. The instability point due to concurrency, above which the responsiveness of the software platforms degrades, strongly depends on the thread model efficiency at any of the above mentioned layers. We set the Tomcat maxThreads parameter to 151, a value that satisfies the Medium Enterprise load. The thread scheduling in all cases was preemptive so that the concurrent client threads can use the CPU time in a round robin fashion.
• **Servlets.** In this implementation we follow the Servlets specification of the multi-threaded model with preemptive scheduling. There is one thread-safe serving object which spawns a new Servlet thread to look after an incoming transaction, adding in this process some delay. When the transaction terminates the thread destroys an event which also causes some latency.

• **RMI.** RMI works with passive code execution. The system is inherently multi-threaded allowing multiple servant objects to run on different ports inside the Java Virtual Machine (JVM). For data integrity and consistency, language level monitors are used on methods to synchronize access to code and data.

• **JAX-RPC** supports the shared thread pool model. The JAX-RPC run time system makes use of a pool of threads, created at start-up, to process the SOAP requests. The thread pool model has the disadvantage of being easily saturated at high arrival rates.

The **connections management** mechanism. This mechanism is responsible for the connections set-up, maintenance and termination at the middleware level and the TCP protocol itself. Connection set-up delays depend on the size of the messages transferred over TCP and the overhead becomes significant when a new connection is created to transfer only a few bytes and then closes.

• **Servlets.** The best practice for Servlets is to buffer information before transmitting it since typical On Line Transaction Processing applications transfer small messages and perform rapid database updates. We configured Tomcat with persistent HTTP 1.1 connections and we set the KeepAlive Timeout interval to 15 seconds. We coded Servlets to be stateful, i.e., to do output buffering before they send packets to the wire. We avoided by this way to set up two connections, one for the user authentication and another one for the debit/credit operation. Liu et al. (2001) offered details of the pros and cons in using HTTP with persistent connections.

• **RMI.** The connections management supported in the RMI transport layer is inherently stateful, i.e., only one look-up of the remote object takes place, while subsequent requests are served on the same network connection, a fact that reduces latencies.

• **JAX-RPC** is inherently stateless in regard to the connections control that it supports. Any new request in a transaction’s cycle opens a new connection to the remote server.

The **socket memory limit** in the JVM. In conjunction with the operating system’s buffering behavior the socket memory is another resource that drives performance. Large messages are always more efficiently accommodated by a large socket buffer space, while there is no need to increase buffering capacity for small sizes. In all the technologies that we tested we configured the socket buffer size at 1MByte.

The combination of the above partial middle-tier processing overheads results in the total software latency $T_{\text{Middleware}}$, that was introduced in equation (2). The total software latency can be now approximately written as:

$$T_{\text{Middleware}} \approx T_{\text{MegHanding}} + T_{\text{MultiThreading}} + T_{\text{ConMgmt}}$$

The experiments of the next Section will reveal the dominating overhead factor in each test case.

9. **Experimental Results**

We took $f$ values in the range of $[0, 1]$ so that each point in the measurements corresponds to a certain type of client server interaction. Any value depicted as response time in the graphs is the mean over three simulation runs.

9.1. **Test Case 1: SOHO Usage Pattern**

In the SOHO case with 10 clients, Fig. 5 demonstrates that the Servlets implementation with persistent connections retained the lowest response times across the whole range that factor $f$ takes.

In more detail, for small message sizes (75kB-250kB), where we expect delays to depend on connection set-up times, Servlets are more responsive than RMI. For larger messages (250kB-750kB) the two technologies perform similarly.
In Servlets we observe a latency reduction around $f$ equal to 0.3 and then a monotonic increase of their response times. RMI times show an earlier and smaller dip, namely in the $f$ range between $[0.06, 0.1]$ and then they start again to increase.

Notably, for high $f$ values in range $[0.9, 1]$ RMI times show a small drop, so that for $f$ equal to unity the response times of Servlets and RMI implementations almost coincide for the first time in the graph.

Web services with JAX-RPC served relatively slowly the SOHO pattern and gave response times that are ten times that of the Servlets times. For messages larger than 150kB the CPU was congested due to the overhead of the text-based SOAP protocol and the processing spent for the necessary XML parsing. In effect, further requests were rejected and response times, even for this light load, exceeded the acceptable values of delays inside the LAN boundaries (25secs-30secs).

It is worth mentioning that Fig. 5 shows that the JAX-RPC response times vary asymptotically with respect to the asymmetry factor $f$. The non-linear dependency of the “waiting time” on the message length coincides with the non-linear dependency of the sojourn time $T_5$ with respect to the arrival rate $\lambda$, as explained earlier in the description of the analytical model. In JAX-RPC the server’s instability point happened around $f \approx 0.2$.

Fig. 6 depicts the cumulative distribution function (cdf) of the application’s response time for the RMI and Servlets implementations. The JAX-RPC cdf is not included in this Figure since JAX-RPC achieved far longer responses. Fig. 6 shows that Servlets statistically performed better than RMI in the SOHO test case.

9.2. Test Case 2: SE Usage Pattern

As Fig. 7 shows, the WWW usage that a Small Enterprise with 50 clients generates causes an increase of the response times in all technology cases. This is a clear indication that the threading systems supported by all technologies start to suffer from the parallel load escalation.

For both Servlets and RMI there is a point of $f$ (value 0.2) around which the response times curve takes a convex shape. This demonstrates the fact that when the data volume increases, times start to be bound by the transmission delays, rather than by the connection set-up overhead.

We observe that for symmetrical scenarios RMI slightly outperforms Servlets for the first time in these experiments.

In the case of SOAP traffic in the SE, Fig. 7 shows that for small message sizes the performance times are very close to the times achieved by JAX-RPC in the SOHO case (Figure 5). The maximum attainable payload by the CPU is almost the same with that of the SOHO case. The CPU served an 88% percentage of the incoming SOAP transactions.
9.3. Test Case 3: SME₁ Usage Pattern

In the case of 100 concurrent clients we observe in Fig. 8 that for $f$ equal to 0.5 (375kBytes uploaded) the RMI and Servlets times are equal. For values of $f$ smaller than 0.5 Servlets outperform RMI, a fact that shows that buffering small and medium size messages is still more efficient than the connections control mechanism supported in RMI.

Servlet performance drops below that of RMI in the second half of the $f$ range. While Servlets increase their times monotonically, RMI exhibits performance improvements. RMI connections seem to operate in the area of heavy messages more efficiently than HTTP connections do. Therefore, what we observed when stressing the CPU with the SOHO workload is now, in the SME case, reversed.

Notably, the RMI response time for $f$ equal to 1 is slightly lower than that with $f$ equal to 0.9.

In the case of the JAX-RPC implementation, according to Fig. 8, performance degrades. The maximum attainable payload size that can now be uploaded is only 75kBytes (server’s instability point at $f$ equal to 0.1). This message size is the half of the messages that the SOHO and SE workloads can upload. We also observe that the JAX-RPC response times under the SME₁ load are close enough to those achieved under the previous SE load.

9.4. Test Case 4: SME₂ Usage Pattern

As shown in Fig. 9, in the case of the workload which models the second SME usage pattern with 150 parallel clients the response times for Servlets and JAX-RPC are larger when compared to their respective times of the previous SME₁ load. It is remarkable that, again, for small values of the asymmetry factor $f$ (below 0.1) the response times are dominated by the connections’ management mechanism supported by each middleware technology and the delays that this imposes.

Fig. 9 shows that, when we increase the load from SME₁ to SME₂, RMI does not exhibit perceptible performance degradation and obviously performs better than Servlets. This can be primarily attributed to the RMI threading system that serves uniformly the parallel requests (RMI makes use of passive code monitors). On the contrary, the numerous dispatches in the Servlet multi-threading system put a burden on the server’s throughput when the contention level is increased.

Fig. 9 shows that the advantage of using persistent connections applies to rapid data update scenarios with moderate data volume communicated. For large values of the asymmetry factor $f$, where transaction
times inevitably increase, the connections set-up overhead increases steadily for HTTP.

Notably, according to both Figures 8 and 9, the worst RMI performance does not occur when we exchange message sizes at the maximum value of \( f \). This trend of the RMI technology was evident in all workload cases. So we can conclude that the RMI implementation capitalizes on the technology’s stateful connection management mechanism, especially in the case of messages above a 675kB size.

Fig. 10 depicts the cdfs for the two competing technologies. The Servlets curve lies now below the RMI curve. Also, the Servlet response times show a higher variation (times up to 4 seconds) than that of the RMI (measured times were up to 2.5 seconds).

In the SME2 test case the JAX-RPC performance degraded even more. Unlike with Servlets and RMI, the JAX-RPC server experienced an early saturation.

Fig. 11 and Fig. 12 that follow depict the RMI and Servlets times for all the four usage patterns across the asymmetry factor \( f \) range. It is demonstrated in Fig. 11 that RMI clients in all usage scenarios perceived a wavy performance with numerous increases and reductions in the application’s response times.

9.5 Throughput performance comparison

The critical bottlenecks that we encountered in the experiments were the server’s processor usage \( CPU_u \) and the Java Virtual Machine’s sockets buffer memory usage \( MU \). We observed that for a given level of simultaneous requests \( N \), there was a threshold value of the arrival rate \( \lambda s \), above which the processor performance degraded with successive transactions failures. According to the java run-time system errors, the requests were rejected due to running out of operating system memory resources, i.e., the overflow of the 1MByte socket buffering space.

With 2.75 incoming transactions per second, the server could successfully process messages with lengths of up to 750kBytes. This size corresponds to an HTML page with one or two medium-size image files embedded.

Fig. 13 depicts the throughput performance of the three technologies under the four workloads.
Fig. 13 shows that Servlets achieved their maximum throughput (3.56Mbps) in the SOHO case for \( f \) equal to 0.3. On the other hand, RMI gave a maximum rate of 2Mbps in the Small Enterprise case, for \( f \) equal to 0.2. Therefore, for the four request types included in the test plan a specific platform’s maximum throughput occurs for different values of the asymmetry factor \( f \).

Regarding the SME1 and SME2 loads the maximum RMI transferring capacity happened for symmetrical and large message exchange between client and server (\( f = 0.9 \) and \( f = 1 \) respectively).

The service look-up mechanism through the RMI remote interface BankingIF had a negative effect on the system’s throughput performance for low and moderate loads. This was even more evident in the case of the JAX-RPC version that made use of a WDSL file to make publicly available the remote interface BankingIFPort.

The overhead caused by the remote reference layer included in the RMI and JAX-RPC middleware stacks justifies why Servlets initially outperformed RMI and JAX-RPC technologies.

However, in high enterprise loads certain stateful RMI features, such as the connections management, the message handling and the threading model improved significantly the application’s response time and the system’s transferring capacity, so that the RMI system finally gave better throughput performance and a tendency to outperform Servlets.

The JAX-RPC version of the server was severely stressed by the SOAP protocol overhead. The JAX-RPC throughput value was approximately one tenth that of Servlets and five times slower than that of RMI. An optimization of the JAX-RPC performance is required.

9.6 Response time comparison

To analyze and correctly interpret the results derived from measurements we averaged the response times in sub-ranges of the asymmetry factor \( f \). These ranges are:

- **SOHO**. We picked experimental data in the \( f \) range \([0.1-0.2]\) and we calculated the mean value that is depicted in Fig. 12. As mentioned earlier, SOHO users usually show browsing activity and they are accustomed to downloading access patterns rather than uploading.
- **SE**. We mean averaged the response times in the \( f \) range \([0.3-0.5]\) as for the SE software usage profile.
- **SME**. For this type we assumed that \( f \) is in the range \([0.6-0.8]\).
- **ME**. We reckoned that the \( f \) range \([0.9-1.0]\) characterizes further a Medium Enterprise’s usage pattern.

An average comparison of the response times is presented in Fig.14. It is shown in Fig. 14 that implementing the banking client server application with Servlets gave better performance in case of the SOHO and SE service demands. As we gradually increased the loading conditions, the Servlets advantage diminished and best performance shifted to RMI.

The advantage of RMI is more evident in higher loads like the SME2 usage type. In the SME2 case RMI performed with roughly one half of the Servlet response times.

Fig. 14 Average response time comparison.

Fig. 14 provides evidence that, especially in scalability terms, RMI is the technology that behaves better amongst the three technologies tested.

The SME1 and especially the SME2 workload enjoyed best responsiveness for the particular test application with a middleware implementation of the distributed object model.

Admittedly, the plain JAX-RPC implementation with no SOAP optimization, such as binary or other compressing/encoding scheme, lacked the performance efficiency required in the synchronous communications model. Experimental evidence discourages plain Web services implementations for time-sensitive applications, especially if they produce symmetrical data traffic of high priority. Thus, to avoid limiting their scope only to asynchronous transfer of lower-priority information, we have to apply some kind of adaptation, such as compression in conjunction with a multicast streaming transport protocol, as suggested by Morse et al. (2004).
Web services could be chosen due to their standards-based interoperability and their flexibility in systems integration.

10. Conclusions and Future Work

Extensive experimentation showed that the distributed middleware platforms alter their behavior when they operate in different loading conditions. We found that the distributed object model, in its Java RMI form, performed efficiently when scaling the loading conditions with a tendency to bounding the response times that the end users perceived.

Nevertheless, the purpose of this work was not to judge the one commercial platform against the other, since the advisability for a middleware option is application dependent. Rather, we aim to offer a performance testing tool able to validly and efficiently, in terms of time and cost, provide a first-cut performance estimation of object oriented distributed models and systems.

In this regard, we intend to further enhance the simulation framework that we used with performance modelling capabilities. Object attributes can hold information related to performance and utilization of the distributed systems resources. Furthermore, the treatment of different service demand classes as actors who possess multiple roles and generate separate activity paths in service execution is an attractive option.

11. References


Pancake, C. M., 1995. The Promise and the Cost of Object Technology: A Five Year Forecast. In:


Christos Tselikis is a Ph.D. candidate at the Department of Informatics at the University of Piraeus, Greece. He received the Dipl. Eng. from the Electrical and Computer Engineering Department of the University of Patras, Greece and his M.Sc. Degree in Communications Technology and Policy, from the Strathclyde University, Glasgow, UK. His working experience spans the Telecommunications and the IT industry. His current research interests include: performance evaluation, traffic characterization, distributed caching strategies, ad-hoc networking.

Sarandis Mitropoulos is a visiting lecturer at the Department of Informatics of the University of Piraeus, Greece, and a System Analyst at a Bank supervised by the Ministry of Economics of Greece. He completed his Ph.D. at the Department of Electrical and Computing Engineering of the National Technical University of Athens (NTUA) in 1994. His Ph.D. dissertation focused on Distributed System and Network Management. He received his degree in Informatics and Computing Engineering for the same department of NTUA in 1990. He has been working in technical and project management and business development in European R&T&D and integrated solution projects, in the areas of system and network management, of advanced telecommunication/telematic services, and of management information systems, as well as of system and network security. He taught in NTUA from 1991 to 1994. He is Senior member of IEEE and member of CNOM and Technical Chamber of Greece.

Christos Douligeris received the Diploma in the Electrical Engineering from the National Technical University of Athens in 1984 and the M.S., M.Phil. and Ph.D. degrees from the Columbia University in 1985, 1987, 1990, respectively. He has held positions with the Department of Electrical and Computer Engineering at the University of Miami, where he reached the rank of associate professor and was the associate director for engineering of the Ocean Pollution Research Center. He is currently teaching at the Department of Informatics of the University of Piraeus, Greece. He has served in technical program committees of several conferences. His main technical interests lie in the areas of performance evaluation and security of high speed networks, neurocomputing in networking, resource allocation in wireless networks and information management, risk assessment and evaluation for emergency response operations. He was the guest editor of a special issue of the IEEE Communications Magazine on “Security for Telecommunications Networks” and he is preparing a book on “Network Security” to be published by IEEE Press/Wiley. He is an editor of the IEEE Communications Letters, a technical editor of IEEE Network and a technical editor of Computer Networks.