JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 5, May-June 2004

A Comparison Framework for Middleware Infrastructures

Apostolos Zarras, University of Ioannina, Greece

Abstract

Middleware is a software layer standing between the operating system and the application, enabling the transparent integration of distributed objects.

In this paper, we propose a framework that facilitates the comparison of middleware infrastructures. Our approach serves for identifying similarities and differences between middleware infrastructures and revealing their advantages and disadvantages when facing the question of choosing one that satisfies the application's requirements. Based on the proposed framework, we compare CORBA with J2EE and COM+, three of the most widely used infrastructures in both industry and academia.

1 INTRODUCTION

Middleware is a current trend in the development of open distributed systems; it stands between the operating system and the application and enables the transparent integration of distributed objects [Bernstein96]. Middleware consists of reusable functionality that offers solutions to frequently encountered problems like heterogeneity, interoperability, security, dependability, etc. This functionality is offered either by the core of a middleware infrastructure, or by complementary services. The former mediates the interaction between distributed objects, while the latter deal with issues like fault tolerance, transactions, naming, trading, security, etc.

In the early 90s, there have been efforts to come up with standards describing the semantics and the structure of middleware infrastructures, capable of supporting a wide range of applications. The CORBA (Common Object Request Broker Architecture) specification [CORBAv3.0.2] is among the most successful results of those efforts. Except for infrastructures that comply with the CORBA standard (e.g., [Henning *et al.98*, Lo *et al.*98, ORBIX, Schmidt *et al.*98, Puder *et al.*00]), there exist others, which are also quite famous and widely used in both industry and academia. Among the most popular, we find J2EE (Java 2 Enterprise Edition) [J2EEv1.4] and COM+ [COM+v1.5].

Given this wide variety of solutions, what is still missing, from a software engineering point of view, is a methodology that facilitates selecting the one that better

Cite this article as follows: Apostolos Zarras: "A Comparison Framework for Middleware Infrastructures", in *Journal of Object Technology*, vol. 3, no. 5, May-June 2004, pp. 103-123. http://www.jot.fm/issues/issue_2004_05/article2

tackles the particular requirements of a distributed application. Recently, the OMG architecture board made a statement concerning the coordinated use of existing standards towards Model Driven Architecture development (MDA) [MDA]. The MDA development process relies on the specification of models of the application's architecture. In a first step, the models are infrastructure-independent (*i.e.*, they abstract away technological details that do not relate to the fundamental functionality of the application). The application's architecture specification may further include a description of technological requirements, which serve for choosing among different infrastructures that may satisfy them. The step that follows consists of refining infrastructure-independent models into infrastructure-specific ones. The MDA development process completes with turning infrastructure-specific models into code.

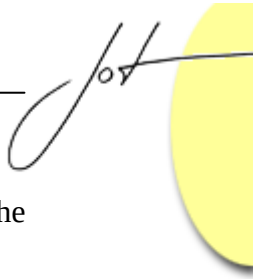
Concerning the issues discussed above, we propose *a framework for the systematic comparison of middleware infrastructures*, which facilitates the first step of the MDA development process. The framework aims at systematically exploring similarities, differences, revealing advantages and disadvantages of middleware infrastructures, when facing the question of choosing one that better satisfies the application's requirements.

The remainder of this paper is structured as follows. Section 2 presents previous work and requirements related to our main objective. Section 3 details the comparison framework. Section 4 presents the framework in action, comparing CORBA with J2EE and COM+. Finally, Section 5 summarizes our contribution.

2 BACKGROUND & REQUIREMENTS

Most of previous approaches to the comparison of middleware infrastructures rely on purely functional criteria. In [Plasil *et al*.98], for instance, the authors compare CORBA, Java RMI and DCOM (a predecessor of COM+) regarding a number of basic concepts (e.g., request/response, remote reference, interface, proxy, marshaling, etc.) and patterns (e.g., the broker pattern, the proxy pattern, etc.), traditionally used for the integration of distributed objects. The overall comparison is faithful from a technical point of view. However, the comparison framework proposed by the authors does not establish relationships between functional concepts, patterns and typical requirements imposed by distributed applications. Hence, given the results from their comparison we cannot reason about which infrastructures are capable of satisfying the requirements of a particular application. Moreover, we cannot reason about which specific concepts and patterns we should use to satisfy those requirements.

In [Roman *et al*.99] and [Gopalan98] the authors also rely on functional comparison frameworks. Comparing middleware infrastructures strictly from a functional point of view can be misleading as it is more than difficult to check all the features of one infrastructure against corresponding features of another one. In general, there is no perfect middleware infrastructure; every one of them has its weak and strong points. Even if a particular infrastructure has less weak points than another does, it may not be suitable



for the specific application that we implement because its weak points may be exactly the ones we have to employ to satisfy the application's requirements.

In [Emmerich00], the author follows a requirement-based approach. More specifically, the author identifies typical requirements of distributed applications over the middleware and examines whether or not different types of middleware infrastructures support those requirements. However, the comparison framework used does not establish the relationship between the requirements and the specific middleware functional concepts we should use to satisfy them.

Based on the issues raised above, we propose a comparison framework which combines both the requirements-based and the functional-based approaches. More specifically, our framework consists of a set of key *requirements* typically imposed by distributed applications over the middleware. Satisfying the requirements depends on the particular *architectural style* (i.e. the different types of elements that can be used for building an application on top of the infrastructure and constraints on the use of those elements) assumed by a middleware infrastructure and the *services* the infrastructure provides. In the proposed framework:

- We define a generic architectural style that satisfies the key requirements; if the particular architectural style of a middleware infrastructure conforms to this generic architectural style, the infrastructure is capable of satisfying the key requirements.
- We identify fundamental services that should be offered by a middleware infrastructure towards satisfying the key requirements.

3 COMPARISON FRAMEWORK

Figure 1 gives the overall structure of our comparison framework, which consists of key requirements imposed over the middleware, a generic architectural style for middleware infrastructures and fundamental middleware services.

Key Requirements over the Middleware

The RM-ODP (Reference Model for Open Distributed Processing) standard [RMODP] discusses the issue of typical requirements on the integration of distributed objects. Moreover, in [Emmerich00] the author further deals with this issue. Based on these approaches, we consider the following requirements:

- *Openness:* The middleware infrastructure should enable extending the applications built on top of it in various ways. (e.g., adding, removing, upgrading, composing services, etc.).
- *Scalability:* The middleware infrastructure should facilitate the effective operation of the applications at many different scales.
- *Performance:* The middleware infrastructure should enable the efficient and predictable, if needed, execution of the applications that are built on top of it.

- *Distribution transparency:* is the property that determines if the application is perceived by users, or developers as a whole rather than as a collection of independent constituent elements. The requirement for distribution transparency is quite generic and it is usually refined into a number of more specific transparencies including:
  - *Access transparency:* the infrastructure should enable accessing local and remote application elements in the same way.
  - *Location transparency:* the infrastructure should enable accessing the application elements without knowledge of their physical location.
  - *Concurrency transparency:* the infrastructure should allow concurrent processing on resources, without interference.
  - *Failure transparency:* the infrastructure should enable service provisioning despite the occurrence of failures.
  - *Migration transparency:* the infrastructure should provide means for changing the location of elements of the application without compromising the application's correct operation, i.e. without affecting the elements that depend on the migrated elements.
  - *Persistence transparency:* the infrastructure should provide means for concealing the deactivation and reactivation of elements from other elements that are using them.
  - *Transaction transparency*: the infrastructure should provide means for coordinating the execution of atomic and isolated transactions.

A Generic Architectural Style

In general, every middleware infrastructure assumes an architectural style that must be followed by applications using the infrastructure. Three basic principles must hold for this architectural style to support the development of *open* and *scalable* applications:

- *Modularity*: The application should consist of a collection of elements, each one providing services, used by the others. Modularity enables the identification of dependencies between the elements that make up the system. Consequently, it allows determining, which elements are affected by the eventual addition, removal or upgrade of services.
- *Encapsulation:* For each constituent element, there is a clear separation between the element's interface and implementation. The interface is a well-defined specification of the provided services, the contract between the element and the entities using it. The implementation is the realization of the provided services. In general, it is safe to change the implementation of an element as long as the element's interface is preserved. Changing an element's interface without compromising the overall application integrity requires that the rest of the application does not depend on this particular interface, at the time of the change.
- *Inheritance:* An interface specification (resp. implementation) may be derived from another one. The derived interface (resp. implementation) provides at least

the services of the base interface (resp. implementation). Inheritance enables the vertical and horizontal composition of services.

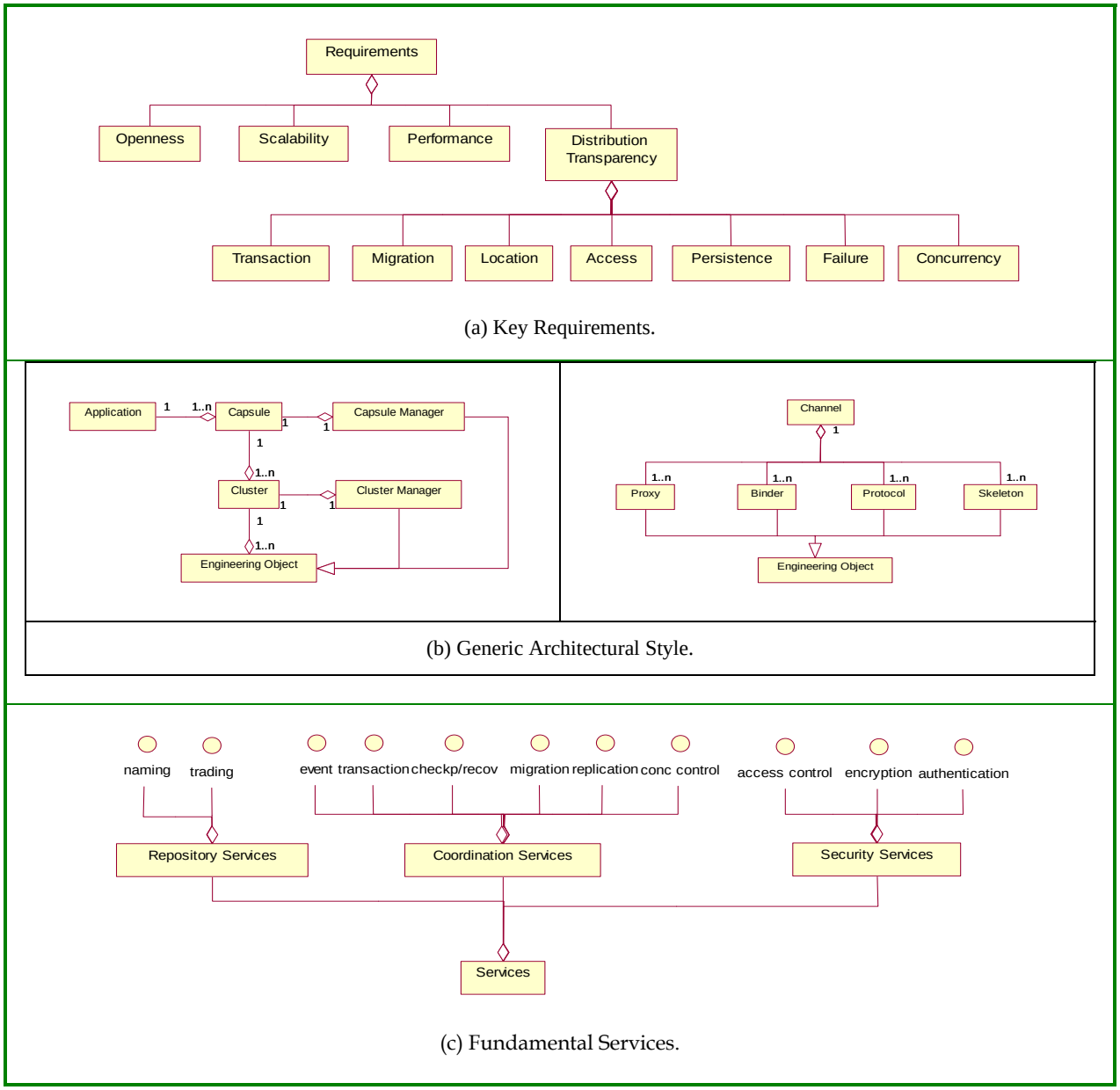


Figure 1: The basic concepts of the comparison framework.

The generic architectural style we assume in the comparison framework respects the aforementioned principles. It further relies on the architectural style proposed in RM-ODP for open distributed systems. More specifically, the basic elements that make up an application are *engineering objects*, i.e. units of data or computation, which we integrate transparently using functionality of a middleware infrastructure. An engineering object can be instantiated multiple times within an application. Instances have state and

collaborate towards the accomplishment of specific tasks. An engineering object provides one or more *interfaces*. Furthermore, an engineering object may require one, or more interfaces. Interfaces may be of the following kinds [RMODP]:

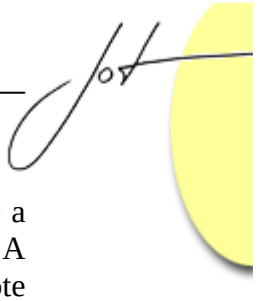
- *Signal interfaces*: defining asynchronous stimuli that can be handled by instances of engineering objects, providing these interfaces.
- O*peration interfaces*: defining operations that can be invoked on instances, providing these interfaces. Invoking an operation causes a request message to be sent by the invoker to the invoked instance. Invoking an operation may further result in a reply sent from the invoked instance to the invoking instance.
- *Stream interfaces*: defining operations that can be invoked on instances, providing these interfaces. The result of invoking a stream operation is the continuous conveyance of information from the invoked instance to the invoking instance.

Following RM-ODP, we assume that engineering objects are organized into *clusters* for the purpose of activation, deactivation, checkpoint, recovery, etc. Each cluster is associated with a *cluster manager*, i.e. an engineering object that coordinates the aforementioned activities. Clusters are organized into *capsules* for the purpose of encapsulation of processing, storage, and request flow. A capsule is associated with a capsule manager, i.e. an engineering object that coordinates the cluster managers of the constituent clusters.

Engineering objects that belong to different capsules communicate through *channels*. More specifically, two or more collaborating objects are associated with a channel, which provides *access transparency*, i.e. it masks differences in data representation and communication mechanisms enabling the inter-operation of the associated objects. A channel is a compound element consisting of proxies, skeletons, binders and protocol objects.

A *proxy* is an engineering object that bridges the semantic gap between local (i.e. elements belonging to the same capsule) and remote elements (i.e. elements belonging to different capsules). Invoking an operation, (or sending a signal) on an object involves holding a *reference* to that object. If both the invoker and the invoked object reside in the same address space, the reference is a typical implementation-language specific pointer (e.g. a C++ pointer). On the other hand, if the invoker and the invoked object reside in different address spaces, the reference is a pointer to a representative of the invoked object (i.e. a proxy) in the invoker's address space. Upon the invocation, the proxy constructs and forwards a request to the remote object, through the rest of the objects that make up the channel. Requests and replies must be converted into a form that is suitable for transmitting over the network. Technically, the previous is achieved through serialization of requests and replies into a byte stream. The serialization procedure is usually called *marshalling*. Several conversions may take place during marshaling, to deal with data representation differences between the invoker's and the invoked object's execution environments (e.g., little-endian, big-endian).

A *binder* is an engineering object that maintains the integrity of a channel (e.g., it monitors communication failures and round-trip times and sets appropriate time-outs; it multiplexes connections to multiple remote objects to optimize resource usage). A



protocol object provides basic communication functionality (e.g., it writes a request to a TCP/IP socket, it performs retransmissions based on time-outs set by the binders, etc.). A *skeleton* is the representative of all objects requiring an interface provided by a remote object, in the capsule of the remote object. The skeleton accepts as input requests built by proxies and uses information that is encapsulated in those requests to perform local invocations on behavioral features (i.e. operations, signals) provided by the remote object. The skeleton may further encapsulate the result of the request into a reply, which is delivered back to the invoker.

Fundamental Middleware Services

So far, we have seen that *openness*, *scalability* and *access transparency* of an application depend on the particular *architectural style* assumed by the middleware infrastructure used for building it. Performance, on the other hand, mainly depends on the realization of the infrastructure's communication *channels.* A basic performance criterion is the communication overhead introduced by the use of the channels. This overhead is usually not negligible; it is the penalty for dealing with communication in distributed applications, executing on heterogeneous execution environments.

Achieving the rest of the requirements, identified at the beginning of this section, relies on the use of complementary services offered by the infrastructure. The use of the complementary services should be *transparent to the application*, whenever possible. More specifically, the infrastructure should provide means that relief the developers from explicitly using complex functionality of the services within the code of the application. The ideal is that developers just setup properties that characterize the objects of the application. Then, the middleware used for integrating the objects implicitly combines functionality of corresponding services to impose those properties (e.g., the developer just sets the replication-style to be active-replication; based on the style, appropriate functionality is used within channels, for multicasting requests sent by clients to groups of replicated objects).

The fundamental services offered by an infrastructure can be divided into three categories: repository, coordination, and security services [RM-ODP].

Repository services provide functionality that allows managing information regarding objects, interfaces, locations, etc. This category includes trading and naming services. A *naming service* defines a name space and provides interfaces through which we associate names with references to objects. Client capsules may then use names to obtain the associated object references. A trading service is a more sophisticated mechanism. The client capsules do not need to know specific names of server objects, they just hold a reference to a trader and use it to request for a required service. The trader maintains a registry that contains references to objects, providing specific services. A client request to the trader is specified in terms of a required interface and additional quality of service properties. The trader looks in the registry for a reference to an object that can fulfill the client's requirements and if there exists one, the trader returns it to the client. Naming and trading services can be used to provide *location* transparency.

Moreover, we can use the trading service to satisfy *scalability* requirements as it can serve as a mechanism for load balancing.

Coordination services include services that can be used to achieve *concurrency transparency* (e.g. locking). *Failure transparency* regarding accidental faults is realized using coordination services for replication, checkpoint and recovery [Laprie85]. Replication may be employed both at the level of engineering objects (use of groups of replicated objects instead of simple objects) and at the level of communication channels (use of request retry and redirection mechanisms). *Failure transparency* concerning intentional/malicious faults relies on the use of security services for authentication, access control and encryption [Laprie85]. *Migration transparency* is achieved using coordination services that allow copying or moving an object from one location to another, without affecting other objects that use it. *Persistence transparency* relies on two key issues: (1) references to persistent objects must remain valid despite the deactivation and reactivation of those objects, (2) the state of the objects must persist to their deactivation and reactivation. In principle, the creation and maintenance of references is a responsibility of the infrastructure. On the other hand, persistent state involves using a complementary checkpoint and recovery service. *Transaction transparency* involves using coordination services that realize atomic commitment protocols (e.g. two-phase commit protocol) and concurrency control protocols that guarantee isolation (e.g. two-phase locking).

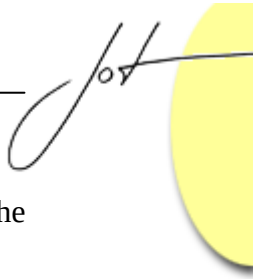
4 THE COMPARISON FRAMEWORK IN ACTION

In this section, we demonstrate the use of the proposed framework for comparing CORBA with J2EE and COM+.

Openness

All three platforms that we assess in this paper support the development of open systems. More specifically, each one of them relies on a particular architectural style, which is aligned with the basic principles and concepts of the generic architectural style we detailed in the previous section.

CORBA forces developers to build applications that comply with the CORBA object model [CORBAv3.0.2]. According to that model, the basic engineering objects are called CORBA objects. Each CORBA object provides a single interface; it is a conceptual entity realized by an implementation language-specific entity (e.g. a C++ object, a Java object, etc.), named servant. In principle, the servant may realize more than one CORBA objects. By default, CORBA interfaces are operation interfaces, as defined in the generic architectural style. An operation may return a reply or not; in the latter case it is called a *one-way* operation. In order to realize a signal interface we have to use the CORBA Event Service [CORBAServices]. Stream interfaces are not supported. The specification of interfaces relies in CORBA IDL (Interface Definition Language), a purely declarative



language that supports interface inheritance. Implementation inheritance depends on the implementation language used for the realization of servants.

CORBA objects that have common properties regarding activation, deactivation, etc. can be organized into clusters managed by a POA element (Portable Object Adapter). Capsules are called CORBA servers. CORBA objects that belong to different servers communicate using channels, which rely on the GIOP communication protocol. A more sophisticated approach for clustering objects that have common properties consists of building CORBA components instead of typical CORBA objects [CORBA-CCM]. Components extend the semantics of simple objects in that they can register to containers, i.e. objects that implicitly manage object activation and deactivation, transactions, security and persistence. To achieve the previous containers combine functionality of standard CORBA services like PSS (Persistent State Service) and OTS (the Object Transaction Service) [CORBAServices]. Technically, each container is associated with a properly configured POA. CORBA components provide more than one interfaces divided into two categories: (1) external interfaces used by other components of the application, and (2) callback interfaces, used by the container towards managing object activation and deactivation, transactions, security and persistence.

OPENNESS	***Architectural Style***							
	Modularity				***Encapsulation***			***Inheritance***
	Engineering Objects	Clusters	Capsules	Channels	Interfaces			
					Operation	***Signal***	***Stream***	
CORBA	***CORBA objects***	✔	✔	***GIOP based***	✔	✔	✘	✔
J2EE	***Java objects***	✔	✔	***RMI based***	✔	✔	✘	✔
COM+	***COM+ Objects***	✔	✔	***DCE RPC based***	✔	✔	✘	✔

Table 1 : CORBA, J2EE and COM+ regarding openness.

J2EE imposes the use of the Java object model [J2EEv1.4] for developing J2EE applications. According to that model, an application comprises a collection of Java objects, each one providing a number of Java interfaces. Java interfaces are operation interfaces. To realize signal interfaces we have to use the Java Message Service (JMS) [J2EEv1.4]. As with the case of CORBA, stream interfaces are not supported. Interfaces are specified using Java language-specific constructs (instead of employing a separate IDL language). Java allows both interface and implementation inheritance. ActivationGroup objects can be used to cluster objects with common properties regarding activation and deactivation. A capsule in J2EE is called a server. Objects that belong to different servers communicate through channels that rely on the Java RMI communication protocol. Alike CORBA, J2EE further provides a more sophisticated approach for clustering objects, which involves building Enterprise Java Beans (EJBs) instead of typical Java objects. EJBs extend the semantics of simple Java objects in that they can register to EJB containers, i.e. objects that systematically manage the

activation/deactivation, transactional processing, persistence, etc. of the registered objects.

In COM+, the basic engineering objects are called COM+ objects and provide one, or more interfaces [COM+v1.5]. COM+ objects are conceptual entities realized by one or more implementation objects written in a conventional programming language like C++, Java, etc. COM+ interfaces are in principle operation interfaces. Signal interfaces can be realized using the COM+ event service [COM+Events]. As with the previous two infrastructures, stream interfaces are not supported. Interfaces are specified using Microsoft IDL (MIDL), a purely declarative language that supports inheritance. Capsules in COM+ are named processes. COM+ objects are organized into clusters, named contexts, regarding common properties, having to do with the objects' activation, deactivation, transactions, security etc. COM+ objects interact through channels that rely on DCE RPC [DCE-RPC].Table 1 summarizes the comparison of the three infrastructures regarding openness.

Scalability

As we detailed previously, the architectural styles assumed by all three infrastructures we examine here support the development of scalable applications. CORBA further provides a trading service that can be used for load balancing [CORBAServices].The CORBA Trader provides a registry of publicly known services. Clients may query the trader for a particular service, providing the specification of the service, in terms of a CORBA IDL interface. Client queries may further include quality of service requirements that must be satisfied by the service provider. The trader answers the clients' queries with references to objects that can successfully offer the required service, while fulfilling the clients' quality requirements.

COM+ supports static load balancing [COM+v1.5]. More specifically, a COM+ system can be configured to assign certain clients to certain servers that execute the same logic. However, the mapping between clients and servers does not change dynamically to reflect changes in the servers' workload. This technique is an easy way for dealing with predictable loads. A more sophisticated solution involves using referral COM+ objects that assign clients to component objects dynamically. J2EE also does not provide a standard trading service [Roman *et al.*99]. However, it provides means for implementing proprietary ones. Table 2 summarizes the comparison of CORBA, J2EE and COM+ regarding scalability.

SCALABILITY	*Architectural Style*	*Trading Services*
CORBA	✔	✔
J2EE	✔	✘
COM+	✔	✘

Table 2: CORBA, J2EE and COM+ regarding scalability.

Performance

Regarding efficiency, several performance evaluation efforts [CORBABench, EJBBench, DCOM] show that the communication overhead is similar in orders of magnitude for COM+, J2EE and various CORBA compliant middleware infrastructures (e.g. OmniORB [Lo *et al.*98], ORBacus [Henning *et al.*98], ORBIX [ORBIX], TAO [Schmidt *et al.*98], MICO [Puder *et al.*00]). In CORBA, we should expect spending from 0.6 to 3.5 ms to send data of a basic type (e.g. long, char, float, double, short etc.) from a client to a server object[1]. J2EE, also seems to be expensive; we should count spending from 4 to 5 ms[1]. Finally, in COM+ 2.5 to 3.5 ms are needed[1]. The communication overhead grows for more complex data types like arrays, sequences, etc.

CORBA is the only one among the infrastructures we consider here that facilitates the predictable execution of applications. In particular, the Real-Time CORBA [CORBA-RT] is an extension of the standard CORBA specification, enabling clients of an application to create priority-banded connections to server objects. Clients can send prioritized requests through those connections. Servers may specify priorities on a per object basis, (e.g., requests targeted to a particular object may have higher priorities compared to requests targeted to other objects encapsulated by the same server). Request processing takes place according to either the client, or the server priority model, depending on specific properties set on the server-side. Clients may further setup timeouts on requests and servers can precise on the number of threads used for request processing. Clients and servers can also customize certain properties of the underlying TCP/IP communication protocol (e.g. the sizes of the communication buffers used). Real-Time CORBA infrastructures come along with scheduling services, facilitating the execution of activities (i.e. sets of requests) according to various scheduling policies (e.g., EDF, rate-monotonic, etc. [Schmidt *et al.*98]). Table 3 summarizes the comparison of CORBA, J2EE and COM+ considering efficiency and predictability.

PERFORMANCE	Efficiency	Predictability
CORBA	0.5 to 3.5 ms	✔
J2EE	3 to 5 ms	✘
COM+	2.5 to 3.5 ms	✘

Table 3: CORBA, J2EE and COM+ regarding performance.

[1] These are representative measures taken from extensive experiments performed in the context of the CORBA comparison project [CORBABench], the EJB comparison project [EJBBench] and [DCOM]. We consider that these measures are sufficient to give an idea of the performance overhead introduced by the infrastructures we examine. An extended performance evaluation of CORBA, J2EE and COM+ is out of the context of this paper.

Access Transparency

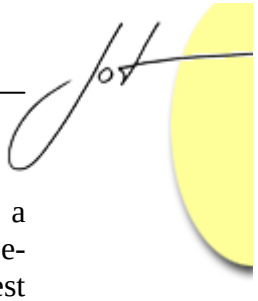
CORBA, J2EE and COM+ channels provide access transparency. More specifically, in CORBA, proxies and skeletons are called stubs and skeletons, respectively. Requests are marshaled into byte streams according to the standard CDR format (Common Data Representation), which deals with byte ordering differences between the machines that host the client and the server object. CDR further deals with the alignment of primitive CORBA data types within messages. Stubs and skeletons are automatically generated using a CORBA IDL compiler.

CORBA binders manage connections according to the GIOP protocol. Connections in GIOP v1.0, v1.1 are asymmetric; the client can issue requests through the connection, while the server can receive requests and send replies but can not issue requests. This restriction is relaxed in GIOP v1.2 and v1.3, where connections are bidirectional. Connection shutdown can be initiated either by a server-side binder, or by a client-side binder. Server-side binders can not initiate connection closure if there exist client requests that are pending. Client-side binders are responsible for multiplexing connections to objects encapsulated by the same server to optimize resource usage. If client-side binders do not support the previous feature, a new connection is created for every server object used by the client. CORBA protocol objects rely on TCP/IP. Other protocol objects may also be used within CORBA channels, as long as they conform to certain transport protocol assumptions specified in the CORBA standard.

ACCESS TRANSPARENCY		Proxy/Skeleton – Marshaling	Binders – Connections				Protocol Objects
CORBA	***GIOP channels***	CDR format	GIOP1.0	GIOP1.1	GIOP1.2	GIOP1.3	TCP/IP
			unidirectional asymmetric	unidirectional asymmetric	bidirectional symmetric	bidirectional symmetric	
J2EE	***RMI channels***	Java Object Serialization Protocol	bidirectional symmetric				TCP/IP
COM+	***DCE RPC channels***	NDR format	bidirectional symmetric				TCP/IP

Table 4: CORBA, J2EE and COM+ regarding access transparency.

J2EE channels rely either on Java RMI [JavaRMI] or on GIOP. According to RMI, at the time when a client obtains a reference, a new proxy is created. Consequently, multiple proxies in the address space of the client may represent the same remote object. Marshaling is based on the Java Serialization Protocol [JavaRMI]. Proxies and skeletons are automatically generated, using the *rmic* compiler. RMI binders manage the opening and closure of bidirectional connections. Moreover, they are responsible for multiplexing connections according to the Java Multiplexing Protocol [JavaRMI]. RMI protocol objects are based on TCP/IP. Other kinds of protocol objects are also supported.



COM+ channels are based on DCE RPC [DCE-RPC]. When a client process obtains a reference to an interface for the first time, a proxy is created. The proxy is reference-counted to avoid creating multiple proxies, representing the same remote object. Request and reply marshalling relies on the DCE NDR (Normal Data Representation) format. Stubs and skeletons are automatically generated using the MIDL compiler. Binders manage the opening, closure and multiplexing of bidirectional connections. Finally, the protocol objects rely on TCP/IP. Other kinds of protocol objects may also be included in COM+ channels. Table 4 summarizes the comparison of CORBA, J2EE and COM+ concerning access transparency.

Location Transparency

CORBA provides both naming and trading services that can be used to achieve location transparency [CORBAServices]. J2EE provides two different naming facilities: a daemon process, called RMI-Registry that realizes a flat namespace and the Java Naming and Directory Interface (JNDI), which is similar to the CORBA Naming service [J2EEv1.4]. The distinctive feature of JNDI over CORBA Naming is that the former generates events, upon request, whenever a namespace changes due to the addition, removal, or update of a name binding. Clients using JNDI can then register event listeners to receive such events. J2EE does not provide, for the time being a trading service.

LOCATION TRANSPARENCY	*Naming Services*	*Trading Services*	*Transparent use of the services*
CORBA	✔	✔	✘
J2EE	✔	✘	✘
COM+	✔	✘	✘

Table 5: CORBA, J2EE and COM+ regarding location transparency.

As in the case of J2EE, COM+ provides naming facilities, but it does not offer any trading service. More specifically, in COM+ there exist two basic global name spaces. The first one contains GUIDs (globally unique identifies), which are bound to COM+ interfaces, or to classes of COM+ objects. The second name space contains names of monikers, i.e. persistent COM+ objects used for storing the state of other COM+ objects.

Using the naming and trading services is typically not a complex task; thus, the middleware infrastructures we assess here do not provide any means to further facilitate it. Table 5 summarizes the comparison of CORBA, J2EE and COM+ concerning location transparency.

Concurrency Transparency

SYNCHRONIZATION TRANSPARENCY	*Concurrency Control Services*	*Transparent use of the services*
CORBA	✔	✘
J2EE	✔	✔
COM+	✔	✘

Table 6: CORBA, J2EE and COM+ regarding concurrency transparency.

CORBA enables the concurrent execution of requests. However, the particular request processing models that can be employed are not precisely defined in the standard. Following, we give typical models that may be supported by CORBA compliant infrastructures:

- The thread-per-request model: a new thread is created for every request delivered to a server.
- The thread-per-client model: a new thread is created for every new client that requests services from a server.
- The thread-pool model: a fixed number of threads are used for serving requests (this model should be provided by infrastructures that comply with the Real-Time CORBA specification [CORBA-RT]).

Concurrency control can be achieved using the CORBA Concurrency Control Service (CCS) - a basic locking mechanism [CORBAServices]. The use of CCS is not transparent to the application; locks should be explicitly acquired and released by the application.

In J2EE, multiple requests may be delivered simultaneously to a Java object. These requests are served in separate threads. Concurrency control is based on standard Java synchronization mechanisms, whose employment can be hidden under the use of the Java *synchronized* clause (if we characterize operations op1, op2 of a Java class with the *synchronized* keyword, J2EE guarantees that upon the concurrent arrival of requests for op1 and op2, op1 shall start after the end of op2, or the inverse).

COM+ allows the concurrent execution of requests in the following sense. COM+ objects within a server are organized into apartments, depending on the request-processing model they use. There are two types of apartments: single-threaded, and multi-threaded. A single-threaded apartment consists of exactly one thread, so requests to COM+ objects that belong to it are served sequentially. A multi-threaded apartment comprises more than one thread, assigned to requests targeted to the objects that belong to the apartment. Concurrency control in multi-threaded apartments can be based on COM synchronization mechanisms (e.g., functionality that realizes the IBlockingLock interface), whose use, however, is not transparent to the application. Table 6 summarizes the comparison of CORBA, J2EE and COM+ regarding concurrency transparency.

Failure Transparency

CORBA was recently extended to support fault tolerance. More specifically, the standard specification for Fault Tolerant CORBA [CORBAv3.0.2] defines basic mechanisms and interfaces for replication, checkpoint and recovery of CORBA object groups. From the client perception, the employment of the fault tolerance mechanisms is transparent since groups are used as normal CORBA objects. Clients hold a group reference instead of an object reference and request services provided by the group.

FAILURE TRANSPARENCY	*Accidental Faults*			*Intentional Faults*	
	Replication Services	*Checkpoint and Recovery Services*	*Transparent use of the services*	*Security*	*Transparent use of the services*
CORBA	✔	✔	✔	✔	✔
J2EE	✘	✘	✘	✔	✔
COM+	✘	✘	✘	✔	✔

Table 7: CORBA, J2EE and COM+ regarding failure transparency.

Depending on the particular replication style used, the CORBA infrastructure either forwards a client request to one member (called primary object) of the group (passive replication), or multicasts the request to all members, while guaranteeing totally ordered request delivery (active replication). In active replication, whenever a member fails, the remaining replicas are used to guarantee correct service provisioning. In passive replication the state of the primary is either periodically stored in a log (*cold-passive* replication), or loaded into one or more backup replicas (*warm-passive* replication). Upon the occurrence of a failure, a backup object becomes the new primary. CORBA further defines mechanisms employed at the level of CORBA channels for request retry and redirection. Fault tolerance support in J2EE and COM+ is limited; channels include mechanisms for network/hardware fault detection and connection recovery.

Regarding intentional faults, all three infrastructures provide security services for authorization, authentication and encryption. The functionality provided by those services is embedded in the communication channels. Hence, the use of security services, in all three cases, is transparent to the application. In CORBA, different kinds of secure channels may be employed, relying on security protocols like SSL, GSS Kerberos and CSI-EKMA (those protocols differ mainly regarding the flexibility they provide in the delegation of identities and privileges; SSL appears to be the least flexible protocol, since delegation is not allowed). Security in J2EE and COM+ is based on SSL channels. However, channels relying on other security protocols (e.g. Kerberos) are also supported [Roman *et al.*99].

Table 7 summarizes the comparison of CORBA, J2EE and COM+ regarding failure transparency.

Migration Transparency

In CORBA there are two possible ways of migrating an object. The primitive way is to invoke an operation on another object that resides at the target location and pass a copy of the object that is to be migrated as a parameter. Then, the original copy can be destroyed. The migrated object must be passed by value. The previous is possible in CORBA only if the migrated object is a *valuetype* (i.e. a special kind of object, whose specification comprises the definition of both the object's interface and the object's state). CORBA channels can redirect requests issued by clients that hold references to the original copy of the migrated object. Consequently, the integrity of the application is preserved. Passing objects by value is also possible in J2EE.

However, this primitive approach to object migration does not deal with dependencies that may exist between a migrated object and other objects. Such dependencies may further impose the migration of the dependent objects. To deal with these cases, CORBA provides a more sophisticated service called CORBA LifeCycle [CORBAServices].

Passing objects as parameters in CORBA and J2EE and using the CORBA LifeCycle service is not transparent to the application. Table 8 summarizes the comparison of CORBA, J2EE and COM+ regarding migration transparency.

MIGRATION TRANSPARENCY	*Passing Objects by Value*	*Migration Services*	*Transparent use of the services*
CORBA	✔	✔	✘
J2EE	✔	✘	✘
COM+	✘	✘	✘

Table 8: CORBA, J2EE and COM+ regarding migration transparency.

Persistence Transparency

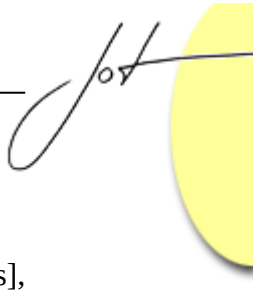
In CORBA, references to an object may be either persistent, or transient depending on the policies of the POA that manages the lifecycle of the object. Moreover, CORBA provides the Persistent State Service (PSS) [CORBAServices], for logging (resp. restoring) objects' states to (resp. from) persistent storage. Checkpoint and recovery of objects' states is not transparent in the sense that it is the responsibility of the application to log periodically the state of its constituent objects. To achieve fully transparent persistence we have to build persistent CORBA components instead of simple CORBA objects and register them into CORBA containers. Upon registration, the containers take over the responsibility of logging and restoring the components' state.

PERSISTENCE TRANSPARENCY	*Persistent References*	*Checkpoint and Recovery Services*	*Transparent use of the services*
CORBA	✔	✔	✔
J2EE	✔	✔	✔
COM+	✘	✔	✘

Table 9 : CORBA, J2EE and COM+ regarding persistence transparency.

In J2EE, object references can be persistent. More specifically, if a client tries to contact an object that can be activated, but is currently not active, the RMI Registry responsible for the object is contacted instead. The daemon shall reactivate the object and provide the client proxy with an updated object reference. Regarding persistent storage, objects can use JDBC or SQL/J to access a database. The use of the previous facilities is not transparent, unless we built EJBs, instead of simple Java objects, and register them to EJB containers that systematically log in database storage the state of the registered EJBs.

References to COM+ component objects are not persistent. The state of component objects can be stored to a database, using ADO or OLE-DB interfaces. However, the use of the aforementioned facilities is not transparent to the application [Roman *et al.* 99]. Table 9 summarizes the comparison of CORBA, J2EE and COM+ concerning persistence transparency.



Transaction Transparency

CORBA comes along with the Object Transaction Service (OTS) [CORBAServices], which realizes the well-known 2-phase-commit protocol. Most implementations of the OTS specification support the execution of both flat and nested transactions. The use of OTS alone does not guarantee isolation. To achieve the previous we have to combine OTS with CCS. The use of OTS and CCS is not transparent to the application. More specifically, application objects that participate in a transaction must register the resources they use to a transaction coordinator. Moreover, before serving transactional requests, the participating objects must try to acquire locks on their resources. When the transaction completes, OTS releases all locks that have been acquired. Hence, the first phase of the well-known 2-phase-locking protocol is a responsibility of the application, while OTS transparently performs the second phase, using CCS functionality.

To achieve fully transparent transactional processing we have to implement transactional CORBA components and register them into CORBA containers. Then, the containers handle client invocations appropriately (e.g., they may implicitly acquire locks, register resources, etc.).

TRANSACTION TRANSPARENCY	*Transaction Services*		*Transparent use of the services*
	Flat Transactions	*Nested Transactions*	
CORBA	✔	✔	✔
J2EE	✔	✘	✔
COM+	✔	✔	✔

Table 10: CORBA, J2EE and COM+ regarding transaction transparency.

J2EE provides a service, named JTS (Java Transaction Service) that is similar to OTS. One significant difference is that JTS only supports the flat transaction model. Moreover, container-managed transactions are supported for EJBs. COM+ provides OLE transactions for the atomic and isolated execution of invocations on COM+ objects. Both flat and nested transaction models are supported. Moreover, COM+ provides MTS (Microsoft Transaction Server), a container for COM+ transactional objects. Table 10 summarizes the comparison of CORBA, J2EE and COM+, regarding transaction transparency.

Overall Assessment

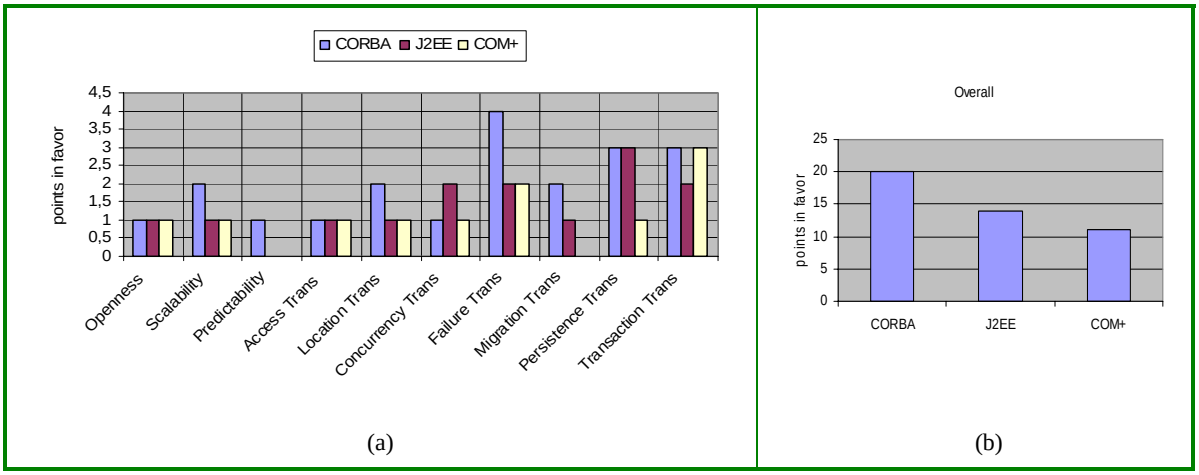


Figure 2: Overall comparison of CORBA, J2EE and COM+, regarding their standard specifications.

Figure 1 gives an overall view of the results we obtained from the comparison of CORBA, J2EE, and COM+. More specifically, we counted one point-in-favor of an infrastructure for every (✔) mark we gave in the detailed comparison (for openness and access transparency we only counted an overall point-in-favor for each infrastructure because they all have common features regarding those properties). Figure 1(a) gives the points-in-favor per-requirement, while Figure 1(b) gives the total number of points-in-favor for each infrastructure. From both figures, we can come into conclusion that CORBA, in general, provides more facilities for satisfying typical requirements imposed by distributed applications. To be fair with the other two infrastructures we have to highlight that this conclusion is based on the comparison of the standard CORBA specification with the specifications of J2EE and COM+. However, not all implementations of the CORBA standard specification provide all of the CORBA services and facilities we identified during the detailed comparison.

Based on the previous remark, we examined a number of available implementations of CORBA (omniORB, ORBacus, ORBIX, TAO, MICO), regarding the availability of the CORBA services and facilities we identified in the detailed comparison. Then, we calculated the points-in-favor for each implementation and compared them with those for J2EE and COM+. Figure 2 gives the results from this comparison. With the exception of omniORB, all CORBA implementations appear better than COM+. However, they are all quite close to J2EE.

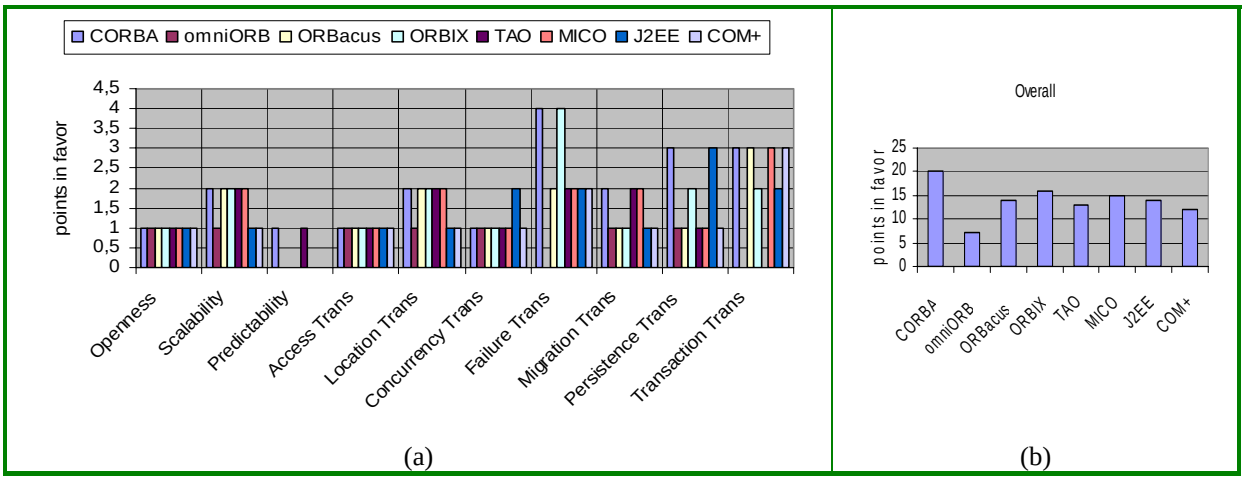


Figure 3: Overall comparison of available CORBA implementations, J2EE and COM+.

5 CONCLUSION

The main contribution of this paper is twofold: (1) it proposes a comparison framework for middleware infrastructures and (2) presents a detailed comparison of three widely used infrastructures in both industry and academia: CORBA, J2EE and COM+. The comparison framework we propose constitutes a foundation for further research we perform in the field of MDA. More specifically, we currently work towards a developer-oriented environment for MDA development, which relies in the approach proposed in [Issarny et al.02]. The core element of our environment is a UML-based representation for the specification of infrastructure independent models of distributed applications. The comparison framework presented here serves for specifying the applications' technological requirements and selecting a middleware infrastructure that provides functionality, which can be used for satisfying those requirements. Our environment further comprises automated procedures for: (1) refining infrastructure independent models into infrastructure specific ones and (2) generating code from infrastructure specific models.

REFERENCES

[COM+v1.5] COM+ v1.5. Microsoft Corporation. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cossdk/htm/pgcontexts_1p0z.asp

[COM+Events] COM+ Event Service. Microsoft Corporation. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cossdk/htm/pgservices_events_5x4j.asp

[CORBAv3.0.2] Common Object Request Broker Architecture (CORBA/IIOP) v.3.0.2. OMG Document, formal/2002-12-06 http://www.omg.org/technology/documents/formal/corba_iiop.htm

[CORBABench] CORBA Comparison Project Web site. Distributed systems Group, Charles University. http://nenya.ms.mff.cuni.cz/projects.phtml?p=cbench&q=3

[CORBA-CCM] CORBA Component Model (CCM). OMG Document formal/2002-06 65. http://www.omg.org/technology/documents/formal/components.htm

[CORBA-RT] Real-Time CORBA v1.1. OMG Document, formal/2002-08-02. http://www.omg.org/technology/documents/formal/real-time_CORBA.htm

[CORBAServices] CORBAServices Specification. OMG Document. http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm

[DCE-RPC] DCE 1.1: Remote Procedure Call. Open Group, 1997. http://www.opengroup.org/onlinepubs/009629399/toc.htm

[DCOM] DCOM Technical Overview. Microsoft Corporation. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cossdk/htm/pgservices_events_5x4j.asp

[EJBBench] EJB Benchmarking Web site. Distributed Systems Group, Charles University, Prague. http://nenya.ms.mff.cuni.cz/projects.phtml?p=ejbc&q=4

[Emmerich00] W. Emmerich. Software Engineering and Middleware: *A Roadmap. The Future of Software Engineering*, ed. A. Finkenstein, pp.117-129 2000.

[Issarny *et al.*02] V. Issarny, C. Kloukinas and A. Zarras. "Systematic Aid for Developing Middleware Architectures". *Communications of the ACM (CACM)*, vol. 45, no. 6, pages 53-58, 2002.

[J2EEv1.4] The Java 2 Enterprise Edition (J2EE) Specification v.1.4. Sun Microsystems. http://java.sun.com/j2ee/

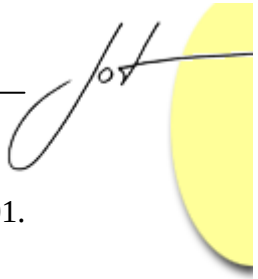
[JavaRMI] Java Remote Method Invocation (RMI). Sun Microsystems. http://java.sun.com/j2se/1.4.1/docs/guide/rmi/spec/rmiTOC.html

[Henning *et al.*98] M. Henning and S. Vinosky. *Advanced CORBA Programming with C++*. Addison Wesley, 1998.

[Gopalan98] S. R. Gopalan. A Detailed Comparison of CORBA, DCOM, and Java/RMI. OMG whitepaper, 1998. http://my.execpc.com/~gopalan/misc/compare.html

[Laprie85] J-C. Laprie. "Dependable Computing and Fault Tolerance : Concepts and Terminology". In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, pp. 2-11, 1985.

[Lo *et al.*98] S. Lo and S. Pope. "The Implementation of a High Performance ORB over Multiple Network Transports". *AT&T Labs Cambridge, Technical Report 98.5*, 1998. http://www.uk.research.att.com/abstracts.html



[MDA] OMG: Model Driven Architecture. OMG Document ormc/2001-07-01, 2001. http://www. omg.org/mda.

[ORBIX] ORBIX v.3 Web site. http://www.iona.com/products/orbix3_home.htm

[RMODP] ISO/IEC. Open Distributed Processing Reference Model (RM-ODP) Part 3: Architecture, 1995.

[Roman *et al*.99] E. Roman and R. Oberg. « The Technical Benefits of EJB and J2EE Technologies over COM+ and Windows DNA". *The MIDDLEWARE Company*, 1999. http://www.middleware-company.com

[Plasil *et al*.98] F. Plasil and M. Stal. "An Architectural View of Distributed Objects and Components in CORBA, Java RMI and COM/DCOM". *Software Concepts and Tools*, vol. 19 no.1, 1998.

[Puder *et al.00*] A. Puder and K. Romer. *MICO and Open Source CORBA Implementations*. Morgan Kaufmann Editions, 2000.

[Schmidt *et al*.98] D. C. Schmidt, D. L. Levine and S. Mungee. „The Design of the TAO Real-Time Object Request Bromer". *Computer Communications*, vol. 21 no. 4, pp. 294-324, 1998.

About the author

Apostolos Zarras got his Ph.D. in the year 2000 from the University of Rennes I, France. From 2000 to 2002, he worked as a research engineer at INRIA-Rocquencourt, France. Currently he is a visiting assistant professor at the University of Ioannina, Greece. His current research interests include model driven architecture development, adaptive middleware, and quality analysis of software systems. He can be reached at zarras@cs.uoi.gr.