MIDDLEWARE: THE LAYER IN BETWEEN

Rui Moreira
Professor Auxiliar
GIMED, Faculdade de Ciência e Tecnologia – UFP
rmoreira@ufp.pt

Álvaro Rocha
Professor Associado
GIMED, Faculdade de Ciência e Tecnologia – UFP
amrocha@ufp.pt

José Braga de Vasconcelos
Professor Associado
GIMED, Faculdade de Ciência e Tecnologia – UFP
jvasco@ufp.pt

Middleware is a widely used term in various areas of distributed systems. This paper focuses specifically on component-based middleware, i.e., the software layer that offers, to application developers, a platform, a programming model and a standard set of services (e.g., persistency, transactions, security, reflection, etc.) capable of hiding the interaction details between components of a distributed system. This middle tier makes language and distribution heterogeneity transparent to developers and decreases the difficulty of managing the creation, distribution, deployment and adaptation of software components.

1. INTRODUCTION

Component models advocate the use of of-the-shelf component solutions that may be combined with distinct vendor's components to produce open applications. By component we mean (cf. ECOOP'96 definition) the "*unit of composition with contractually specified and explicit context dependencies only*" which can be composed and deployed independently by different users (Szyperski, 1998). Other authors consider components as binary executables and distinguish them from subroutines or open libraries that can be modified at source code level (Krieger, 1998). These software blocks are usually integrated in a overall *component model* that defines a programming model and an associated platform supporting a set of services (e.g., naming, transactions, persistency, security) and tools (e.g., IDEs, compilers) that hide implementation details and promote the *component-based development* (CBD) paradigm (Brown, 1997). This middle layer (cf. middleware) enables the creation and reuse of pre-fabricated, parameterised and replaceable units already tested and bug free (Szyperski, 1998). Such approach reduces the learning curve and the software development costs and, at the same time, increases the quality and interoperability between enterprise systems. Currently there are three major players in the market of *enterprise component models*, i.e., those designed for distributed systems:

- Object Management Group (OMG): an industrial consortium that is trying to bring some international convergence into a standard distributed architecture, initially with the CORBA standard (OMG, 1996), (Orfali, 1998) and now with the *CORBA Component Model* (CCM) initiative (Cobb, 2000);
- Sun Microsystems: the major server vendor that firstly introduced a client-side/desktop component model (JavaBeans) (Sun, 1997) and then a server-side/enterprise component model (Enterprise JavaBeans) (Roman, 2002); both propose a Java based approach to distributed applications;
- Microsoft Corporation: the major desktop operating system vendor that primarily introduced a Windows-based binary solution to software development (COM+) (Goswell, 1995), (Box, 2000) and that currently adopts/defends a service-oriented approach based on a cross-language framework (.NET) (Microsoft, 2001).

This paper presents the OMG standard approach and then the two vendor positions that are becoming *de facto* component standards. For each component model we present an historical overview and then the architecture and the essential technical details. The paper finalizes with a comparison between the presented component models.

2. OMG VIEW OF MIDDLEWARE (CCM)

2.1. HISTORIC OVERVIEW

The *Object Management Group* (OMG) is an extensive consortium responsible for the *Object Management Architecture* (OMA), an open middleware specification for building interoperable component-based applications. More specifically, this reference model describes the *Common Object Request Broker Architecture* (CORBA) and proposes both a standard for middleware infrastructure and a programming model for assembling and deploying distributed applications (Emmerich, 2000), (Mowbray, 1997). CORBA requirements evolved over the years as reflected in the release of three main versions of the standard. The first version defines a distributed object model that separates interfaces from implementations. Interfaces are defined in a language and platform independent notation (cf. *Interface Definition Language* - IDL) that is mapped by a specific compiler to a concrete programming language and platform. Distribution is then supported by the use of a standard communication bus (cf. *Object Request Broker* - ORB) that handles the transparent invocation of methods (*marshal* and *un-marshal*) on local or remote objects (Orfali, 1998). Moreover, CORBA specifies a common set of services and facilities that help the development of distributed applications by integrating mechanisms for naming, event communication, life-cycle management, etc. More recently, CORBA version 2 focused on ORB interoperability and object activation management by standardising the *Internet Inter-ORB Protocol* (IIOP) and specific *Portable Object Adapter* (POA) policies. The use of an IDL compiler combined with the runtime platform (ORB) manages cross language, cross platform and cross location independence, while the TCP/IP-based inter-ORB protocol assures cross vendor interoperability. Finally, the CORBA version 3 adopted in 2001, standardises the *CORBA Component Model* (CCM) which extends the features and services that enable the implementation, configuration, assembly and deployment of distributed component-based applications (Marvie, 2000). The first two CORBA versions tackle interoperability through a distributed *object model* whereas the last version standardises a full component model. The former address the distribution of applications as sets of objects serving well-defined public interfaces while the latter increases the levels of integration and flexibility by automating some tedious and error prone tasks that are usually delegated/solved by developers in *ad hoc* ways (e.g., deploy and install implementations, activate and configure services, life-cycle management). The main features of the former object model were only briefly outlined above since our main focus is on the details of the component-model.

2.2. COMPONENT MODEL ARCHITECTURE

The CCM is a server-side component model that may be used to assemble and deploy multilingual components. CCM standardises and automates the component development cycle

(from specification to deployment) by defining a middleware infrastructure and a set of support tools. This component model architecture permits us to define the interfaces supported by the components, automate their implementation and pack the components in assembly files (e.g., JAR, DLL) that are automatically deployed on server hosts. The architecture uses proven design solutions (cf. design patterns) that enable the automation of code generation and associated usage of a container infrastructure that mediates the component access to the system services for handling security, transactions, events and persistency (Cobb, 2000). CCM focuses on the provision of system services required by server applications and that are implemented by the container, freeing the application code from complex and error prone tasks thus allowing developers to concentrate on the business logic details. CORBA proposes the use of multiple interface inheritance to address the extension of functionality but, since it does not allows overloading, this mechanisms has proved to be a rather limited approach. CCM complements inheritance with the provision of several interaction mechanisms generically called *ports* that control three aspects (see Table 1) of the component's structure and functionality: assembly (interface wiring), loosely coupled interaction (publish/subscribe pattern) and configuration (customisable properties) (Marvie, 2000), (Merle, 2002). Ports provide new ways to use and combine components, i.e., they enable the configuration of bindings, subscribe or publish events and customise component attributes. Through this mechanism it is possible to replace one component by another (with new interfaces/functionality) without affecting older clients, hence supporting simple component evolution. Moreover, the standard prescribes several meta-interfaces to introspect, compose and configure components.

Aspects	Ports	Description
Composition (components wiring)	facets	provided interfaces exposing the component functionality (via synchronous or asynchronous operations).
	receptacles	required interfaces (references/connections to other components) representing external dependencies on other component's facets.
Asynchronous communication (loosely coupled interaction)	event sources	channels used to publish (exclusive provider channel) or emit (broadcast shared channel) events.
	event sinks	channels used to consume (asynchronous) events from other components or notification sources.
Configuration (customisation)	attributes	transient or persistent attributes used to parameterise component instances; exception mechanisms control attribute values.

Table 1 **structural aspects of CCM components.**

Inter-component communication is just one of the aspects of interaction since components must also run in (and interact with) heterogeneous environments. Therefore, to get access

to the underlying infrastructure (e.g., POA policies, CORBA services) the CCM introduces an interaction pattern based on containers (see Figure 1). A *container* is automatically generated for each component implementation and constitutes the component's view of the surrounding system (Ruiz, 2000). The container shields components from the details of the platform and provides a framework (standard runtime API) for seamless and automatic integration of core services (Cobb, 2000). The container provides a series of uniform interfaces (cf. *internal interfaces*) establishing the context of interaction with the environment (e.g., *stateless, conversational, durable*), i.e., the interfaces through which the component accesses the system services (e.g., transaction, security, persistency, notification). The type of the internal interfaces depends on the component category (cf. *service, session, process* and *entity*). The container is also responsible for using hook methods (cf. *callback interfaces*) to notify the component of certain events (e.g., persistency, transaction) (Wang, 2000). The component implements and registers these callbacks in the container to be notified of the events.

For each component type there is an associated *home* component that is responsible for attributing primary keys and instantiating components. Furthermore, the component container uses an activation framework (e.g., *ServantActivater*, *ServantLocator*) that exploits the

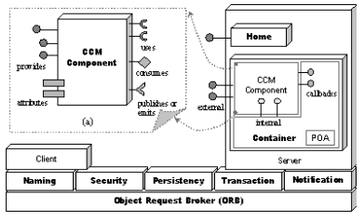

Fig.1 **CCM architecture and container model; (a) CCM component structure.**

POA mechanisms to control servant's lifetime (e.g., activation, deactivation, lookup) according to a chosen policy (cf. servant lifetime policy). This way it is possible to control (depending on the component category) the *activation* and *passivation* of components on a per-method, per-transaction, per-component (via specific callbacks) or per-container basis, in co-operation with the persistence service. Along with the lifetime policies, the CCM also standardises management policies that determine the way containers handle (on component's behalf) transactions, security, events and persistency. The container intercepts the requests from clients and, according to the requirements (declared in the component XML configuration file), enables and executes pre-processing strategies (e.g., activation, transaction, persistence, pooling, caching) before delegating requests to the component.

2.3. DEVELOPMENT, ASSEMBLY AND DEPLOYMENT

The CCM deployment architecture uses an activator daemon that is responsible for on-demand activation of the component server, respective container and associated home (Ruiz, 2000). Home instances are registered in a centralised database and may be located through a *HomeFinder* interface (similar to the name service). Additionally, CCM defines a packaging technology for deploying multilingual binary executables. Developers have to implement the component's functionality and leave the non-functional aspects to be automatically generated from declarative descriptions (Marvie, 2000). CCM supports the development process with automated mechanisms to generate the necessary runtime container requirements (Ruiz, 2000). Specifically, the CCM *Component Implementation Framework* (CIF) defines a set of *Application Programming Interfaces* (API) and tools that automate the code generation of several management strategies (e.g., life-cycle, transaction, security, event and persistence policies). This framework automatically exposes different aspects of the implementation that may be embedded in a component's implementation (Wang, 2000). CCM also standardises a declarative language called the *Component Implementation Definition Language* (CIDL), which is used to describe component's implementations and associated persistent states. A CIF compiler reads the component's CIDL description and generates default component behaviour (e.g., introspection, activation, state management). The resulting implementations are called *executors* (e.g., facets, home, container) and provide hook methods that may be used by developers to add custom behaviour and adapt the default implementation (Wang, 2000).

The CIDL compiler is also responsible for generating component *descriptors*, which are XML-based files used to define the component category (e.g., entity, session), component features (e.g., ports), policies applied to the container (e.g., lifetime, transactions, security, events and persistency) and segmentation (independent units). CCM defines several XML-based descriptor files (cf. *component descriptor, software package descriptor, assembly descriptor and property file descriptor*) which conform to the WWW Consortium's *Open Software Description* (OSD) *Data Type Definition* (DTD). Component segments and descriptors are joined in a *package file*, i.e., a archive file that contains one or more implementations of a component and the associated XML description files. Component packages may be installed or grouped with other packages in an *assembly file*. Descriptor files are used at deployment-time to automatically create and configure the POA hierarchy and resolve component dependencies.

2.4. SUMMARY

The OMA standard addresses several structural and functional concerns (e.g., communication, composition, life-cycle, serialisation) but also tackles design and runtime interoper-

ability by defining information modelling mechanisms (component assembling and deployment description facilities) and a detailed (though burdensome) programming model. This approach is supported by specific compilers that map component specifications across languages and platforms and also automate software development with the generation of customised executors and glue code (e.g., component interfaces, factories, container policies, callback interactions). CCM is generically more open than other concurrent standards but suffers from lack of concrete and ready to use market implementations. Furthermore, CCM compliant environments do not guarantee a scalable functioning system since it is still the responsibility of the designer to implement and check if the architecture requirements are accomplished. These architecture concerns and additional issues related with the flexibility and openness will be analysed in chapter 3 along with reflection as a generic approach to adaptation.

3. SUN VIEW OF MIDDLEWARE (JAVABEANS/EJB)

3.1. HISTORIC OVERVIEW

The *JavaBeans* specification, from Sun Microsystems, defines a software component model for Java applications running on desktop computers (Sun, 1997). A *bean* is a reusable software component that can be manipulated visually in a builder tool, which differentiates it from class libraries that cannot benefit from visual manipulation even if providing the necessary functionality. Some of the unifying features of JavaBeans are the support for property customisation (control the appearance and programmatic behaviour of beans), event handling (communication metaphor based on delegation and event listeners), persistence (serialisation of beans state for later reload or transmission over a network) and introspection (analyse and use the beans internal structure, e.g., properties, events, methods and exceptions). Important enhancements were introduced with the Glasgow specification which extends JavaBeans with new user interface mechanisms (e.g., drag-and-drop, activation framework) and support for containment and provision of services (Sun, 1998). Specifically, the new extensible runtime containment and services protocol proposes an abstraction entity representing the environment called *bean context* which is used to logically group a set of related beans (*child beans*) in a hierarchy of contexts. Furthermore, the bean context enables the dynamic addition of services that may be discovered and used by child beans. Therefore, the Glasgow specification standardises not only the way beans can be nested (e.g., add, remove) into a context but also how to iterate or enumerate child beans (contained beans) and discover and use services registered on the context by service providers. Contexts use the Java event model to register listeners and deliver events to notify beans about added or removed beans and services. Bean contexts standardise an interoperable infrastructure that provides seamless integration with the runtime environment regardless of the underlying platform (Stal, 2000).

JavaBeans defines a component model for Java whereas *Enterprise JavaBeans* (EJB) standardises a framework model for distributed Java components (Jubin, 2000). JavaBeans are *development components*, i.e., small-grained application bits that we may use to build larger-grained components or applications (Roman, 2002). However, JavaBeans are not *deployable components*, i.e., they do not have a runtime environment (cf. container) where to live and that isolates them from the particularities of the underlying system. For that reason, Sun released in 1998 the EJB specification for deployable components (cf. enterprise beans) whose architecture will be detailed below. Some similarities with CCM will be noticed since EJB was the seed model for CCM basic components which (contrary to CCM extended components) do not offer the extensive set of ports functionality.

3.2. COMPONENT MODEL ARCHITECTURE

EJB defines a component architecture using a scalable environment (based on containers) that provides runtime services for managing component activation, concurrency, security, persistency and transactions (Jubin, 2000). EJB defines a component model by standardising the contracts (cf. *context* and *callback* interfaces) and services offered by the runtime environment and the patterns

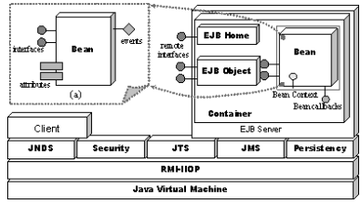

Fig.1 EJB architecture and container model; (a) Bean structure.

of interaction between components (see Figure 1). Additionally, EJB specifies three types of beans: *Entity, Session and MessageDriven*. *Entity beans* represent business elements that embody data and are by nature transactional (and persistent); these beans may handle the persistency themselves (bean-managed persistence) or delegate it to the container (container-managed persistence). *Session beans* can be used to model business processes in a transactional and secure manner without the need for persistent storage. *Message-driven beans* are created by containers to asynchronously handle messages from the *Java Messaging Service* (JMS) sent, for example, to a queue transparently associated with the bean (Stal, 2000).

Depending on the bean type, developers must implement pre-determined interfaces (e.g., *SessionBean, EntityBean, MessageDrivenBean*). These callback interfaces are used by containers to manage and notify the beans about certain events (e.g., bean activation or passivation, instance removing, transaction completion, etc.). Moreover, each type of

bean expects a specific interface or context from the container (e.g., *EntityContext, SessionContext, MessageDrivenContext*) for getting entity bean's primary key, identifying the bean caller, transaction demarcation, etc. These contracts and communication patterns increase the integration with the container and the interoperability between components therefore promoting off-the-shelf component reuse and faster development cycles. Furthermore, the container provides a uniform interface with services such as naming (*Java Naming and Directory Service*), security (public/private key authentication and encryption), transactions (based on *Java Transaction Service* or OMG *Object Transaction Service*) and messaging (*Java Messaging Service*) (Stal, 2000), (Roman, 2002).

Containers act as a layer of indirection between clients and bean instances. Each container provides a network-aware object (cf. *EJB object*) that exposes the bean functionality and *intercepts* every method call before delegating it to the bean. This EJB object is an automatically generated instance with container-specific embedded knowledge (code) about bean activation, transactions, security and networking (Roman, 2002). Each EJB object implements the *remote* interface that enumerates all business methods exposed/implemented by the respective bean.

Some of the particularities of beans come from the close dependency on the Java language, which directly determines their security, persistency, openness and portability. Java is compiled to an intermediate form of code (cf. *bytecode*) and interpreted by a *Java Virtual Machine* (JVM) therefore making beans portable across different computer architectures and operating systems though language dependent. Java, similarly to other object-oriented languages, separates interfaces from implementation, i.e., developers may define the interface of a component (cf. component contract) and then provide different implementations for the same interface for tackling implementation evolution. Clients may use interfaces instead of classes (cf. interface-based programming) because at compile time the classes are not needed, therefore it is possible to change the component's logic without interfering with the client's code (Roman, 2002). The component implementation is not tied up with the client code because the interface hides it but, in contrast to CORBA and COM, this is only possible between a client and a component both written in Java. Furthermore, beans can implement one or more Java interfaces (multiple interface inheritance) but only single implementation inheritance (eliminating the diamond inheritance problem) and can be made persistent (*entity beans*) or non-persistent (*session beans*).

3.3. DEVELOPMENT, ASSEMBLY AND DEPLOYMENT

The EJB specification defines the processes and contracts established for component's assembly and deployment. An EJB component implementation is provided in a enterprise bean class which conforms to certain rules and well defined interfaces (exposing certain aspects) that differ according to the component type (cf. session, entity, message). For bootstrapping purposes, the container uses a container-generated factory (cf. *home object*), which implements the remote home interface (cf. *EJBHome*) and is used by clients to create and find EJB objects. The deployment of beans is made with *ejb-jar* files that pack bean classes, remote interfaces, home interfaces and bean properties files together with a unique XML-based deployment descriptor containing information on all the beans (e.g., home name, bean class name, primary key, container fields, etc.) and their dependencies. *Deployment descriptors* declare middleware services needed by components (e.g., life-cycle policy, persistency handling, transaction control), avoiding non-standard format *manifest* files (previously used with *JavaBeans*) (Szyperski, 1998).

Furthermore, EJB relies heavily on the *Remote Method Invocation* (RMI) platform to support dynamic class loading, automatic activation, remote exceptions and distributed garbage collection (Szyperski, 1998). RMI is a distributed architecture, which uses a RPC-based protocol supporting inter-process communication (Emmerich, 2000). For example, *EJBHome* and *EJBObject* remote interfaces rely on the RMI infrastructure for transparent distribution of component's functionality. The *Java Development Kit* (JDK) includes a set of classes and development tools (e.g., rmi compiler) to support the automatic generation of distribution classes (e.g., stubs), glue code (e.g., container policies) and XML-deployment descriptors, thus alleviating the developers efforts and responsibilities.

3.4. SUMMARY

EJB specifies a standard distributed architecture for building component-based business applications. EJB builds on the previous JavaBeans client-side component model and particularly on the Java language which enables the WORA principle (*Write Once Run Anywhere*) though making it language dependent (Stal, 2000). EJB also relies on the RMI architecture that, in contrast to CORBA, is not an open standard though supporting (via serialisation) objects passed by value. Moreover, RMI simplifies and automates the development and distribution process and an integration effort was made to make EJB portable to CORBA systems, particularly through standard connectors (vendor specific bridges that link different architectures) along with RMI-IIOP mappings (Szyperski, 1998). Finally, the EJB software engineering process (similarly to CCM) is grounded on a set of tools and code generators that automate the development and deployment process by hiding the cumbersome details of handling distribution and component management policies (e.g., life-cycle, security, transactions, persistency).

4. MICROSOFT VIEW OF MIDDLEWARE (COM+ & .NET)

4.1. HISTORIC OVERVIEW OF MICROSOFT PREVIOUS STYLE (COM+)

In 1991 the first version of OLE (*Object Linking and Embedding*) provided a standard way to *embed* or *link* data objects (e.g., text, graphics, images, sound, video, etc.) inside document files, hence permitting us to easily create and manage compound documents (Brockschmidt, 1996). In 1993 the second version of OLE evolved the compound document paradigm and provided a generic infrastructure (cf. component model) to support desktop component-based development (cf. ActiveX). The core architecture behind this infrastructure is the *Component Object Model* (COM) which provides a binary solution to interoperability and extensibility (Goswell, 1995). In the middle of the 90's the Windows NT4 introduces *Distributed COM* (DCOM) which supports inter-process communication across distributed machines through an RPC-based wiring protocol called *Object RPC* (ORPC) (Pattison, 2000). More recently, the component model was extended (COM+) and integrated with Windows 2000 for supporting the development, configuration and administration of distributed systems with automatic and (Windows-based) integrated control over several aspects of business applications (e.g., security, synchronisation, transactions, queues and events). The next subheadings focus on the architectural and technical aspects of COM and COM+.

4.2. COMPONENT MODEL ARCHITECTURE

The COM component model extends object-oriented design principles by hiding the component's implementation behind its interfaces (cf. *encapsulation*) and allowing components to be replaced by different implementations of the same set (or super set) of interfaces (cf. *polymorphism*) without the need to recompile their clients (Emmerich, 2000). These principles are possible because component services (collection of interfaces) are separated from their implementation through an indirection mechanism called a *virtual table* (cf. C++ *virtual functions or vtables*) (Szyperski, 1998). At runtime, an interface is just a typed pointer (cf. *Interface Identifier* - IID) to a named table that references the functions (methods) implementing the services exposed by the interface. This *binary interface structure* allows the interoperability between software components written in different languages as long as compilers can translate language structures into this binary form (Goswell, 1995).

Independent development brings together naming conflicts between interfaces and their implementations, hence developers must assign a different *Interface Identifier* (IID) and *Class Identifier* (CLSID) to each new specified interface and class implementation, respectively. An interface is considered immutable and even a simple variation requires the generation of a new IID that uniquely identifies the new version of the interface. These *global unique identifiers* (GUIDs) are 128 bit identifiers (cf. OSF-based *Universal Unique Identifiers*) used as runtime types. Source level names are then mapped to these GUIDs that are unique in time and space.

COM forces clients to dynamically query components and find out their functionality (supported interfaces) before calling any service. Components expose their interfaces through the *IUnknown* interface (mandatory implemented by all objects) which clients use at runtime (via *QueryInterface* method) to find out the supported interfaces. The COM infrastructure also uses *IUnknown* to update reference counting and perform object lifetime management (cf. automatic garbage collection) (Brockschmidt, 1996). Therefore, two techniques may be used to extend a component's functionality. A common design solution is to create a new interface and include it in the implementation object, which will support the extra functionality. The other solution is to extend the existing interface and share the same *vtable* to reduce memory overhead. Both solutions provide a new component that may keep being used by legacy applications and reused by new applications (that take advantage of the new functionality).

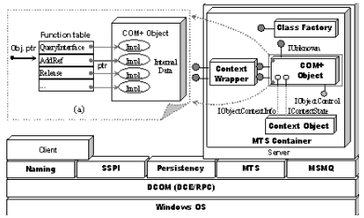

Fig.1 COM/COM+ architecture; (a) virtual table architecture.

COM separates the interfaces of components from their implementation while COM+ separates component behaviour from it's state (Stal, 2000), i.e., provides an environment (runtime platform and services) that exposes (in a declarative way) some aspects (attributes) of COM objects (e.g., activation, deactivation, security, transactions). Furthermore, COM+ supports an *attribute-based programming* model which depends heavily on the *interception layer* (cf. container) created by the runtime. This layer acts as a hook (cf. *context wrapper*) used to execute *pre* and *post* system processing actions, for example, on methods marked to be executed in the scope of a transaction. For each COM+ object, the container automatically creates a shadow *context object* (implementing predefined interfaces, e.g., *IObjectContextInfo, IContextState*) that controls and provides information about the object (e.g., instantiation, transaction and security management). Furthermore, each COM+ object is notified about certain events (e.g., activation, deactivation) via the *IObjectControl* interface which may implement certain developer-programmed actions.

4.3. DEVELOPMENT, ASSEMBLY AND DEPLOYMENT

Component interfaces are defined with the *Microsoft Interface Definition Language* (MIDL) that is an OSF/DCE-based extension of IDL. Then the MDIL compiler generates marshalling classes and type information (e.g., proxy, stub, header files, type library) needed to accomplish binary compatibility, i.e., deployment of components developed in different languages across several environments. Distribution is grounded on the DCOM communication architecture (proxy/stub RPC communications) that generically runs on top of the Windows operating systems and is supported by a set of tools (Horstmann, 1997). COM does not support inheritance though basic component composition is available through containment (create instances of inner objects and reuse their services transparently to external clients), delegation (wrappers that insert behaviour before or after delegating the method calls to inner classes) and aggregation (expose the inner interfaces through the outer *IUnkown*) (Goswell, 1995). Even so, no extended composition mechanisms are supported to allow architecture awareness.

COM+ exposes certain service attributes in a declarative way (as opposed to the imperative API-based style of programming). The installed components are registered in a system catalogue (cf. registration database - *RegDB*) that keeps a profile of configurable attribute settings for each component. Developers set the values of the exposed attributes (via Windows graphical tools) and then the runtime environment interprets the values and implements the associated actions (e.g., controlling concurrency and lock management, load balancing and data consistency) (Chappell, 1998), (Box, 1998). This approach provides a runtime configuration alternative to personalise the middleware. For example, we may set up declarative security checks through roles (in security profiles) or mark a component to be executed in the scope of a transaction (which then is automatically and transparently managed by the *Microsoft Transaction Service* (MTS) on every method invocation).

4.4. HISTORIC OVERVIEW OF MICROSOFT CURRENT STYLE (.NET)

More recently Microsoft launched the .NET component model (pronounced *"dot net"*) which constitutes a new paradigm for building and delivering software as a service (Microsoft, 2001). This new platform uses several Internet standards (e.g., XML, SOAP, WSDL) to describe and expose component services through their properties, methods and events. Moreover, it allows the deployment of loosely coupled services across Internet firewalls (via SOAP/HTTP) and the integration of different client gadgets (e.g., PCs, PDA, cellular phones) though deeply dependent on the operating system set of enterprise services (e.g., security, transaction monitors, message queuing).

4.5. COMPONENT MODEL ARCHITECTURE

The .NET framework permits us to write applications and expose their services using different programming languages. This is accomplished through a *Common Language Runtime* (CLR) which is a unified runtime environment, i.e., a canonical execution engine that is able to integrate components written in different languages (see Figure 2). The CLR engine loads components based on their types (type-based loader) and permits us to access their services through type coercion operations (type-based resolution mechanisms) (Box, 2000). Moreover, the runtime engine combines a *Common Type System* (CTS) with extensive usage of component *metadata*. CTS sets the rules for defining and using types across different languages while *metadata* provides a uniform mechanism for storing and retrieving information about types, hence supplying the foundations to accomplish multilingual integration. Additionally, .NET provides a *Common Language Specification* (CLS) that describes a set of language features (e.g., primitive and composite types, natural-size types, references, exceptions) and rules for using these features (e.g., defining, creating, binding and persisting types). This specification expresses a set of naming and designing guidelines for mapping features between different languages (Microsoft, 2001).

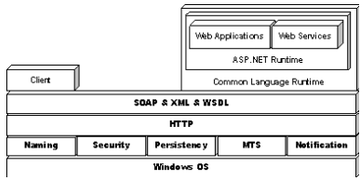

Fig.2 NET architecture and runtime engines.

The .NET framework is complemented by a set of unified class libraries for standard programming (e.g., I/O, math, etc.), accessing the operating system services (e.g., network, thread, cryptography, etc.), debugging code and building enterprise services (e.g., transactions, events, messaging, etc.). These libraries include a set of classes called ASP.NET, which are tailored to the development of Web-based applications. ASP.NET provides an infrastructure with a set of controls that simplify both the server side (web forms that mirror the typical HTML user interface, e.g., buttons, list boxes, etc.) and client-side programming (check client capabilities and choose the appropriate interface). The ASP.NET infrastructure also includes an HTTP runtime (different from the CLR) which is an asynchronous and multithreaded execution engine that processes HTTP commands. The HTTP runtime uses a pipeline of HTTP modules that route the HTTP requests to a specific handler (managed .NET class). This modular architecture supports addition of services to applications by supplying more HTTP modules (e.g., security, state, management) or handlers (one per URL) in the pipeline (cf. variant of the chain of command design pattern (Gamma, 1994)).

4.6. DEVELOPMENT, ASSEMBLY AND DEPLOYMENT

Components are compiled into a CPU-independent language called *Microsoft Intermediate Language* (MSIL) that is targeted to the CLR engine (cf. managed code). The CLR is able to recognise and execute *portable executable* (PE) files, which are image files that combine MSIL code with *metadata* (stored in metadata tables and *heaps*). This approach avoids the need for multiple and disparate metadata formats (e.g., type libraries, header and IDL files) and enables the use of reflection, serialisation and dynamic code generation in a type safe manner (Meijer, 2002). MSIL compilers are responsible for automatically emitting metadata into the PE file (e.g., information describing types, members, references, inheritance, etc.). The runtime environment then uses this binary metadata information (cf. managed data) to locate and load classes, control memory usage, resolve invocations, manage runtime context boundaries, enforce security and compile to a particular computer architecture by using specific *just-in-time* (JIT) compilers. Metadata is the .NET language neutral way to provide binary information describing: assemblies (e.g., unique identification, dependencies on other assemblies, security permissions), types (e.g., base classes, implemented interfaces, visibility), members (e.g., methods, fields, properties, events) and attributes (extra metadata modifying the properties of types and members).

The .NET framework uses *assemblies* as the fundamental unit of composition and deployment. An assembly contains the collection of types and resources that form a logical unit of functionality and deployment (cf. component) and can be roughly compared to EJB JAR files (Meijer, 2002). Contrary to COM components, assemblies do not require registration in the operating system. The assembly sets the scope for references and type resolution, security permissions, version control and deployment. These issues are treated on a per assembly basis through the information stored in the *assembly manifest* (e.g., version, file dependencies, exported types) associated with each assembly PE file. Moreover, service data and functionality may be programmatically exposed using XML-based meta-information (i.e., platform independent data type description) attached to each component. Programmers can expose web service methods through metadata tags that force the automatic generation of one proxy class per-method. The full description of components enables the CLR to dynamically assemble cache information about installed components and avoid using the registry (to categorise and locate components) thus simplifying component installation.

4.7. SUMMARY OF COM+ AND .NET

The COM+ component model focuses on the provision of enterprise distributed applications while .NET addresses the programming services (cf. language services) for Web-based software development (Microsoft, 2001). COM+ takes domain-neutral aspects out of the source code and exposes them through declarative attributes that can be used to control service contexts and transaction support (e.g., process synchronisation, security profiles, automatic transactions) (Box, 2000). Nevertheless, no additional attributes are exposed hence making this mechanism limited for the majority of COM developers. Additionally, COM+ type information management is rather cumbersome, i.e., it uses disparate information formats (e.g., IDL, type libraries, MIDL-generated strings embedded in proxy DLLs) sometimes with no mappings between them. Furthermore, COM+ runtime type information (cf. type library) permits us to advertise only the types exported by a component but not component dependencies (Box, 2000). The new paradigm for service development (.NET) uses self-describing components (assemblies) to tackle these limitations. Each .NET assembly sets the scope for type names (avoiding per-type GUIDs) and explicitly represents component dependencies. Moreover, assemblies avoid the separation of disparate meta-information sources because the metadata is automatically compiled into the image PE file. Finally, .NET type information is extensible (via system attributes), can be applied to different elements (e.g., classes, methods, properties) and is available at runtime via reflection (more on reflection in (Moreira, 2003)). Developers may then use these attributes (cf. keywords and settings) that transparently integrate with the COM+ attribute-based context and transaction infrastructure.

5. CONCLUSION

Current enterprise component models propose an integrated container-based environment for automating the management of transactions, security, persistency and event notifications, though CCM (contrary to EJB and COM+) is not tied up to a particular language (like Java) nor operating system (like Windows). CCM was designed closely to the EJB specification and these component models can be considered conceptually equivalents (Ruiz, 2000). Both support different types of components which automatically determine the available container interfaces and the policies for managing the component's state and persistency (component-managed or container-managed). Furthermore, CCM and EJB define three approaches to demarcate transactions (cf. client-managed, container-managed and server-managed) while COM+ supports only automatic transactions (MTS-managed). Moreover, COM+ defines only one type of component which provides a simpler programming model but, with limited expressiveness and deeply dependent on the MTS environment. Despite being more difficult and complex to learn and manage, CCM and EJB architectures may be considered more flexible and open than COM+ which builds on top of the operating system services. Nevertheless, COM+ is a binary standard that allows the integration of several languages without compromising the performance. Another significant aspect is the recurrent use of meta-information for describing (though with limitations) the structure and behaviour

of components. Meta-information is widely used in CCM (e.g., interface repository), EJB (e.g., bean descriptors) and COM+ (e.g., type libraries) but is particularly visible in .NET where the metadata is embedded in the image files and then extracted using reflection to reason about the system and control assembly, enforce security, manage serialisation, perform compiler optimisations, etc. The combination of meta-information and reflection is an interesting topic addressed in (Costa, 2001) for managing type evolution. The use of reflection (and inherently meta-information) to reason about the system and validate middleware reconfigurations are further explored in (Moreira, 2003).

REFERENCES

Box, D. (1998). The Third Wave. *In: MSDN, Microsoft Systems Journal*, January.

Box, D. (2000). *House of COM: Is COM Dead? In: MSDN Magazine*, December, URL: http://msdn.microsoft.com/msdnmag/issues/1200/com/default.aspx (July 2005).

Brockschmidt, K. (1996). What OLE Is Really About. *In: MSDN Library*, July.

Brown, A. (1997). Background Information on CBD. *In: SIGPC*, Vol. 1, No. 18, August, http://www.jorvik.com/technical/CBDIntro.html (September 2003).

Chappell, D. (1998). How Microsoft Transaction Server Changes the COM Programming Model. *In: Microsoft Systems Journal*, January, URL: www.microsoft.com/msj/0198/mtscom/mtscom.aspx (July 2005).

Cobb, E. (2000). CORBA Components: The Industry's First Multi-Language Component Standard, BEA Systems, OMG Meeting CCM Tutorial, Oslo (Norway), June, URL: http://www.omg.org/cgi-bin/doc?omg/00-06-01 (July 2005).

Costa, F. (2001) *Combining meta-Information management and reflection in an architecture for configurable and reconfigurable middleware*. PhD thesis, Lancaster University.

Emmerich, W. (2000). *Engineering Distributed Objects*, Chichester, UK, John Wiley & Sons.
Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading Mass, Addison-Wesley Professional Computing Series.

Goswell, C. (1995). The COM Programmer's Cookbook. *In: MSDN Library*, September, URL: msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomg/html/msdn_com_co.asp (July 2005).

Horstmann, M., Kirtland, M. (1997). DCOM Architecture. *In: MSDN Library*, July, URL: msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomarch.asp (July 2005).

Jubin, H., Friedrichs, J. (2000). *Enterprise JavaBeans by Example*. New Jersey, Prentice Hall.

Krieger, D., Adler, R. (1998). The Emergence of Distributed Component Platforms. *In: IEEE Computer*, March, pp 43-53.

Marvie, R., Merle, P. (2000). From CORBA to Components. *In: 14th European Conference on OO Programming*, June, URL: corbaweb.lifl.fr/papers/index.html#00_ECOOP (Sept. 2003).

Meijer, E., Gough, J. (2002). Technical Overview of the Common Language Runtime. White paper, MS Research, URL: http://pag.lcs.mit.edu/reading-group/meijer01clr.pdf (July 2005).

Merle, P. (2002). CORBA Component Model Tutorial, LIFL/INRIA, Yokohama OMG Meeting, April 2002, URL: http://www.omg.org/cgi-bin/doc?ccm/2002-04-01 (July 2005).

Microsoft Corporation (1996). DCOM Technical Overview. *In: MSDN Library*, November, URL: http://msdn.microsoft.com/library (July 2005).

Microsoft Corporation (2001). Microsoft .NET Framework Reviewers Guide. *In: MSDN Library*, 2001, URL: http://download.microsoft.com/download/VisualStudioNET/Utility/7.0/W9X2K/EN-US/frameworkevalguide.doc (July 2005).

Moreira, R. (2003). *FORMAware: Framework Of Reflective components for Managing architecture Adaptation*. PhD Thesis, Lancaster University. URL: http://www2.ufp.pt/~rmoreira (July 2005).

Mowbray, T., Malveau, R. (1997). *CORBA Design Patterns*. Wiley Computer Publishing.
Object Management Group (1996). *Common Object Request Broker Architecture Specification (CORBA/IIOP)*. URL: http://www.omg.org/technology/documents/corba_spec_catalog.htm (Sept. 2003).

Orfali, R., Harkey, D. (1998). *Client/Server Programming with Java and CORBA*. 2nd Edition, Wiley Computer Publishing.

Pattison, T. (2000). *Programming Distributed Applications with COM+ and Microsoft Visual Basic 6.0*, 2nd Edition, Microsoft Press.

Roman, E. (2002). *Mastering Enterprise* Javabeans, 2nd Edition, John Wiley & Sons.

Ruiz, D. (2000). CORBA & Components. *In: V Jornadas de Ingeniéria del Softarware y Bases de Datos*, Valladolid, November. URL: http://www.ditec.um.es/~dsevilla/ccm (Sept. 2003).

Stal, M. (2000). Component Technologies for the Middle Tier: CCM, EJB, COM+. Siemens AG - Corporate Technology. In: OOPSLA 2000, Minneapolis, 2000, URL: http://www.stal.de/Downloads/middletier_components.pdf (July 2005).

Sun Microsystems (1997). JavaBeans API Specification, Version 1.01, 1997, URL: http://java.sun.com/products/javabeans/docs/spec.html (July 2005).

Szyperski, C. (1998). *Component Software: Beyond Object Oriented Programming*, Addison-Wesley, ACM Press.

Wang, N., Schmidt, D., O'Ryan, C. (2000). Overview of the CORBA Component Model. White paper, September. URL: http://www.cs.wustl.edu/~schmidt/PDF/CBSE.pdf (July 2005).

ACKNOWLEDGMENTS

Rui Silva Moreira was supported by FCT (programme POCTI - III EU Communitary Supporting Framework) and was also supported by UFP.