Distributed Systems Middleware

David Andersson, 810817-7539, (D)
Rickard Sandell, 810131-1952, (D)

EDA 390 - Computer Communication and Distributed Systems
Chalmers University of Technology
2005-04-30

Table of Contents

Introduction

The word “*middleware*” is now days very popular for use in technical texts. Only a few years ago this term could hardly be found in any computer science dictionary, except for the most recent publications, and a search on the Internet would only find a couple of thousand web pages that mention it. But, today a search for “middleware” using one of the most popular search engines like *Google.com* generates millions of hits. This indicates an explosive increase of interest for this technology.

Unfortunately, “middleware” is also frequently used as a catchword and sometimes even in the wrong context. Therefore, it can be confusing and difficult to figure out the meaning of this acronym. This report will focus on explaining the concept of middleware (in the area of distributed application development) and give examples of different types of middleware. Finally, a comparison between some popular software implementations of middleware will be made.

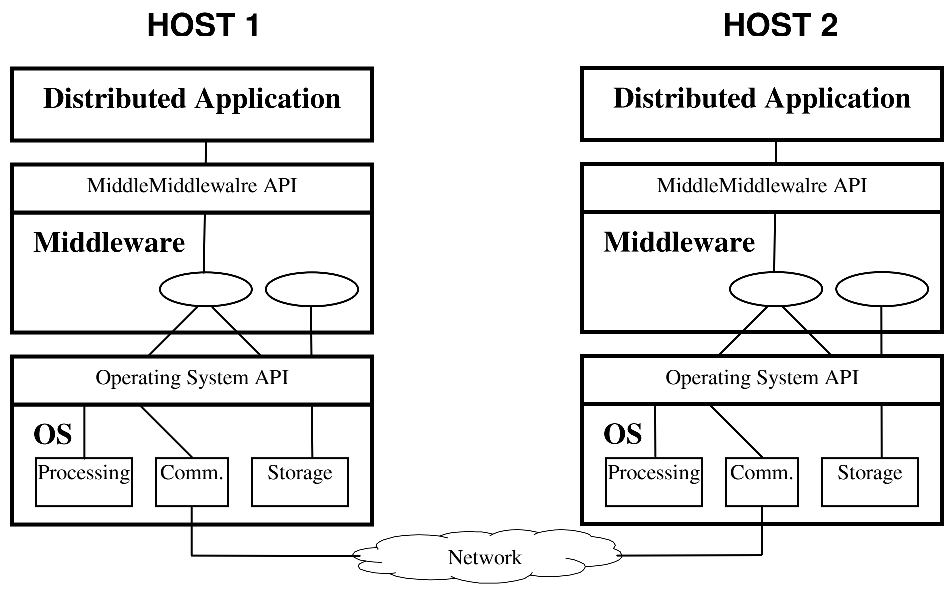
What is middleware?

Middleware is often informally referred to as “plumbing” because it connects different parts of a distributed application with data pipes and then passes data between them. This explanation is not incorrect but it doesn’t describes the key idea with middleware either.

Basic network communication functionality is offered by the operating system that runs on a computer and it can usually be programmed as it is, for example by using the client/server concept of the *TCP/IP* protocol stack. However, that type of programming is not trivial and can be very error-prone. In addition, it is specific to the platform and architecture that is used and it doesn’t support the full complexity of object oriented programming that builds many of today’s applications.

Middleware is a class of software technologies that is designed to solve exactly these problems by hiding the complexity and heterogeneity in distributed systems. It is defined as a layer of software above the operating system but below the application program, thereby providing a higher-level programming framework than *Application Programming Interfaces* (APIs) such as sockets. This reduces the burden on programmers and gives more flexibility (depending on the middleware being used).

Hardware and network heterogeneity is always masked by a middleware but most frameworks also mask heterogeneity of operating systems or programming language, or both. A few, like *CORBA*, even hide heterogeneity among vendor implementations of the same middleware standard.



Classification of middleware

Middleware is a wide concept and the definition is general enough to classify a big variety of software solutions as middleware. Thus, many ways of categorizing them is possible. One of them is the *Gartner Model*, which divides middleware into three categories:

1. *Presentation Middleware* only cares for displaying data remotely. A Web browser and server communication via the HTTP protocol (hypertext transfer protocol) can for example be classified into this category.

2. *Application Middleware* is used to distribute application logic and functions to create a general purpose programming platform for distributed applications. Its goal is to enable programmers to build interacting components without having to hardcode system specific details. For instance, this type of middleware can give the illusion of working with local objects when requests to remote resources are made.

3. *Database Middleware* is deployed to access database management systems remotely. SQL requests sent to a DBMS and transferring results back to the client is a typical task for database middleware.

General definition of middleware:

A software layer between the operating system and application that enables the interaction of potentially distributed application components, reaching a certain degree of transparency and independence from surrounding runtime environments.

This text will primarily concentrate on Application Middleware but most of the principles can probably also be applied to presentation and database middleware. Further, Application Middleware systems historically rely on the two basic approaches *Remote Procedure Call* (RPC) and *Distributed Computing Environment* (DCE) that will be explained in the next chapter.

Application Middleware

RPC (Remote Procedure Call) was the first widely accepted middleware solutions for programming of distributed applications. The earliest implementation of RPC is from 1984 but the most popular one was shipped by *Sun* in 1985. *Sun RPC* was delivered as a part of their *Open Network Computing* (ONC) package. One of the main applications of RPC is Sun's *Network File System* (NFS).

DCE, which is based on RPC, introduced a new concept of hierarchically structured *services*. Although DCE had some major limitations, the idea of services was important and serves as a foundation for the modern middleware described later in this report.

Remote Procedure Call

The purpose of RPC is to make remote procedure calls look syntactically identical to local calls. The main advantage with this approach is easy parameter handling that allows for static type checking at compile time, a feature that is not given with pure socket communication.

Unfortunately, a syntax similar to local procedure calls doesn't imply identical semantics in a distributed system. In other words, the main drawback with RPC is that it doesn't take into account that semantics can differ between hosts due to runtime environments, address spaces and process scheduling. The result of this is that it is not possible to pass pointers using RPC and therefore this method is not suitable for fully object oriented solutions.

Implementations of RPC are based on descriptions of the interface using a predefined RPC language, from which *stubs* and *skeletons* for the client- and server-side are generated with an RPC compiler. These stubs and skeletons take over the task of packing and transporting the parameters on booth sides, and is of course using underlying protocols like *TCP* or *UDP*.

Finally, RPC is also accepted as a standard (RFC 1831) by the *Internet Engineering Task Force* (IETF).

Distributed Computing Environment

DCE is a development of the *Open Group*. Its purpose is to provide a distribution platform for client/server applications. This is done by defining a set of layered services, so that higher level services can always use lower level services. The defined services, starting with the highest level, are:

- *RPC Service* - client-server communication.
- *Security Service* - authentication, authorization, account management.
- *Directory Service* - single naming model throughout the environment.
- *Time Service* - synchronization of system clocks in the network.
- *Threads Service* - enables execution of multiple distributed threads.
- *Distributed File Service* - provides access to files across a network.

The Threads Service is based on the POSIX standard 1003.1c for lightweight processes which makes it possible to build advanced distributed systems that makes use of creating/deleting, manipulating and synchronizing threads. Together with the other service layers this makes DCE suitable for many different kinds of general purpose tasks.

However, the major limitation with DCE is that it relies on RPC as its only communication mechanism, and its non-object-oriented design (it now also exist object-oriented extensions to DCE). Nevertheless, an important contribution of DCE is the concept of splitting middleware functionality into a set of services and its decentralised features. These ideas can be found in many modern middleware platforms.

Message-Oriented Middleware

Message-Oriented middleware, *MOM,* takes charge of relying data between two applications by putting it in a message queue that can be accessed over the network. This solution is really nothing more than a generalization of the well-known procedure of sending and retrieving an e-mail (using *SMTP* and *POP3*). This asynchronous middleware communication is mostly used for simple one-way exchanges of data where timing is not an issue.

Distributed Object Middleware

Distributed Object Middleware provides the abstraction of an object that is remote but whose methods can be invoked just like those of a local object. Distributed objects support all benefits of object-oriented programming techniques like encapsulation, inheritance and polymorphism. Thus, using this kind of middleware in development of new distributed applications is the most general approach.

On the other hand, object-oriented middleware are complex software. Even if the programmer doesn't need to learn a new programming language, but can define interfaces which can be programmed in any language supported by the middleware, it still takes a great deal of time to master this kind of software. Therefore, in some cases it might be more appropriate to choose a simpler (but more limiting) implementation using RPC or MOM instead.

Middleware, and especially object-oriented middleware, is often used to integrate legacy components. That is, the middleware interface can be programmed to translate requests between systems that originally were not able to communicate because of legacy problems.

The *Common Object Request Broker Architecture* (CORBA) is a standard for distributed object computing, and is by many considered to be the broadest in terms of scope. Other important implementations of object-oriented middleware are Microsoft's DCOM and Java RMI. These will be described and briefly compared in the following chapters.

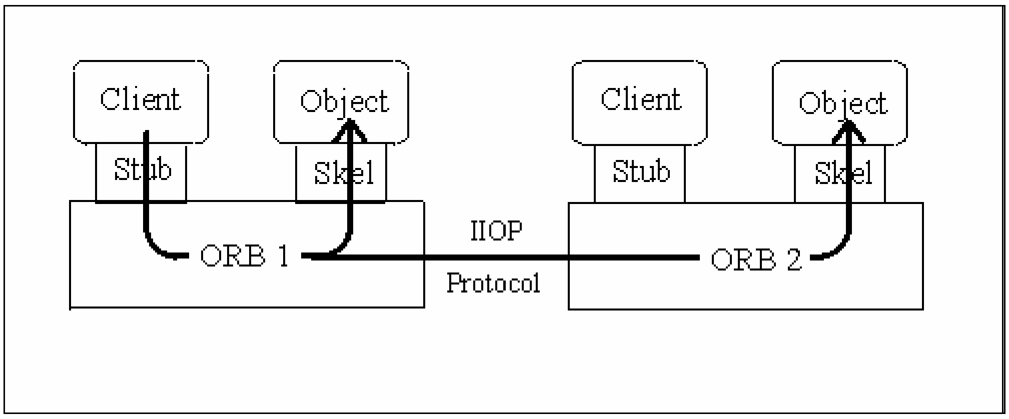
Overview of CORBA, DCOM and Java RMI

CORBA

CORBA, *Common Object Request Broker Architecture* is a standard for working with objects over a distributed environment. It can be seen as an object oriented variant of RPC. The standard is registered by the members of *Object Management Group* (OMG).

The central part of CORBA is the *Object Request Broker* (ORB). The ORB works as a central Object Bus where each CORBA object can interact with other CORBA objects. The interaction is transparent and the objects can be located either locally or remotely on another host. The ORB supports search for methods, converting formats, rearranging parameters and synchronization. The communication between ORBs is done via the *Internet Inter-ORB Protocol* (IIOP).

Each CORBA server object has an interface and exposes a set of methods. To request a service, a CORBA client acquires an object reference to a CORBA server object. The client can now make method calls on the object reference as if the CORBA server object resided in the client's address space. The ORB is responsible for finding a CORBA object's implementation, preparing it to receive requests, communicate requests to it and carry the reply back to the client.



A CORBA object interacts with the ORB either through the ORB interface or through an *Object Adapter* - either a *Basic Object Adapter* (BOA) or a *Portable Object Adapter* (POA).

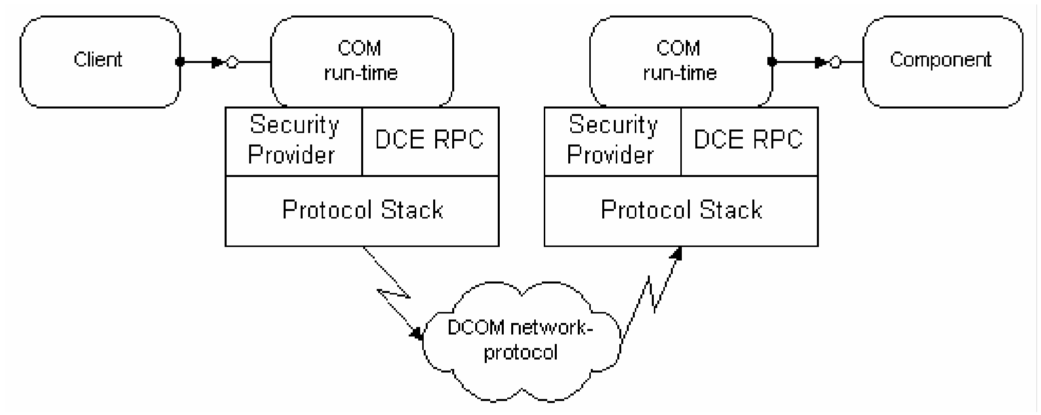
According to the CORBA specification, an object adapter is responsible for the following functions:

- Generation and interpretation of object references
- Method invocation
- Security of interactions
- Object and implementation activation and deactivation
- Mapping object references to the corresponding object implementations
- Registration of implementations

To define object interface the *Interface Definition Language* (IDL) is used. This makes it possible for objects written in different languages to communicate with each other. For each object that is specified with IDL a *Stub* is generated. The Stub is linked to the client at compilation and works as the client's picture of the server. The equivalence on the server side is called a skeleton. Both the client and the server use an interface called the *ORB interface* when they want to use the methods that the ORB supplies. A static connection between server and client in CORBA is fast but it has the drawback that every time the server interface changes the clients has to be recompiled. Therefore CORBA can also be used in a dynamic connection. In a dynamic connection the *Dynamic Invocation Interface* (DII) and the *Dynamic Skeleton Interface* (DSI) is used. The DII and DSI are used to create interfaces dynamically during runtime instead of creating them at compilation. Since CORBA is just a specification, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is an ORB implementation for that platform.

DCOM

DCOM, *Distributed Component Object Model* is an extension of Microsoft's *Component Object Model* (COM). COM defines how components and their clients interact. This interaction is defined such that the client and the component can connect without the need of any intermediary system component. DCOM extends the protocol so that the component and the client don't have to reside on the same computer.



DCOM runs the protocol *Object Remote Procedure Call* (ORPC) to support the communication between two machines. The ORPC is built on top of DCE RPC and interacts with COM's run-time services. A DCOM server is a code segment that that can serve objects of a particular type at runtime. Each DCOM server objects can support multiple interfaces each interface represents a different behavior of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces. The client object is now able to call the server object's exposed methods through the acquired interface pointer as if the server object resided in the client's address space.

To define interfaces for the DCOM and RPC objects the *Microsoft Interface Definition Language* (MIDL) is used. MIDL supports two different forms of interface descriptions, the basic COM interface *IUnknown* and the OLE automation interface *IDispatch.* Every DCOM object must implement the *IUnknown* interface. The *IDispatch* interface is an extension of *IUnknown* and can be seen as a gateway to many more interfaces. Type information of the objects is stored in *type libraries* (.tlb) created by the MIDL compiler. The type libraries are used to dynamically invoke objects implementing the *IDispatch* interface. A *Unique Universally Identifier* (UUID) is used to uniquely identify every class and interface in COM.

Since the COM specification is at binary level it allows DCOM server components to be written in different programming languages. The hardware platform must support COM services in order to provide DCOM.

Java RMI

Java RMI, *Java Remote Method Invocation* is a standard developed by JavaSoft. RMI uses a protocol called the *Java Remote Method Protocol* (JRMP). It is dependent on *Java Object Serialization* to transmit objects as a stream. This means that both RMI server object and client object must be written in Java. Every RMI server object defines interfaces to be used for accessing the object outside the current *Java Virtual Machine* (JVM) from another JVM that can be located on another machine. The server machine has an *RMIRegistry* where it holds information about the available server objects and provides a naming service for RMI. A RMI client acquires an object reference to a RMI server object by doing a *lookup* for a Server Object reference and invokes methods on the server object as if the RMI server object resided in the client's address space. The server objects are named using URLs and when the client wants to acquire an object it specifies it by using the URL address.

When a Java/RMI client requests a service from the Java/RMI server, it does the following:

- Initiates a connection with the remote JVM containing the remote object,
- Marshals the parameters to the remote JVM,
- Waits for the result of the method invocation,
- Unmarshals the return value or exception returned, and
- Returns the value to the caller.

The use of serialization means that both data and code can be passed between a server and a client. This also means that different instances of an object can run on both server and client machines.

Because Java RMI relies on Java it can be used on many different operating system platforms as long as there is an existing implementation of JVM for that platform.

Comparison of CORBA, DCOM and Java RMI

Programming Languages

Since CORBA is a specification it is not dependent on only one language. Any language can be used as long as there is an ORB implementation for that language. DCOM is a binary standard and supports multiple programming languages. Java RMI on the other hand can only be used as middleware between clients and servers that is implemented in Java. This is of course a major drawback with Java RMI compared to CORBA and DCOM.

Definition of Interface

CORBA and DCOM are very much alike in this area. They both offer an *interface definition language* (IDL) for describing interfaces for their respective objects. The IDL is in both cases a neutral language used to define mappings between different programming languages, although there are some differences in semantics and interface notations. Java RMI is different though it doesn't have an IDL. Instead it has defined interface declarations as a separate concept in the language and the interfaces are stored as .java files. A Java object is accessible by remote Java clients if it implements the *java.rmi.Remote* interface.

Object Oriented Support

DCOM and Java RMI both have support for multiple interfaces for an object. Each interface represents a different view or behavior of the object. Earlier versions of CORBA did not have this feature but it is now implemented in the later versions.

DCOM server objects can create several object instances of multiple DCOM object classes depending on the number of interfaces being used. An object in CORBA or Java RMI on the other hand is served by one server object instance which can represent many other object instances.

Interface Identification

In DCOM interfaces are uniquely defined by the UUID which is registered in the Windows system registry. UUIDs make it possible for a server to extend the functionality by implementing a new interface with a new UUID. CORBA uses the interface name to identify an interface and the implementations by mappings in the *Implementation Repository*. Java RMI identifies classes by name and implementations by mappings to an URL in the RMI registry.

Supported Platforms

This is a weakness in DCOM because it is mainly supported on the Windows platform. Attempts to run DCOM on other platforms has not been very successful. CORBA and Java RMI does not have this weakness, they are supported on nearly all platforms.

Günter Rackl,
Monitoring and Managing Heterogeneous Middleware, January 2001.

Derek Slater, *Middleware Demystified*,
http://www.cio.com/archive/051500_middle.html, May 15 2000.

Distributed Computing Environment (DCE) overview,
http://www.opengroup.org/dce/info/papers/tog-dce-pd-1296.htm, April 2005.

David E. Bakken, *Middleware*, Washington State University, 2003

Andrew T. Campbell, Geoff Coulson, Michael E. Kounavis,
Managing Complexity: Middleware Explained, IT Pro magazine October 1999.

Gopalan Suresh Raj
A Detailed Comparision of CORBA, DCOM and Java/RMI

Carl-Fredrik Sörensen
A Comparision of Distributed Object Technologies

Sven-Arne Andréasson, Christer Carlsson
Distribuerade Databehandlingssystem