F.O.A.M.
Free Object Access Method

NET Services

Documento: NET Services.doc
Versione: 0.40.2k30204
Autore: Mario Meo – Colombo

Introduction: What's Middleware?

Traditional compute environments are of heterogeneous type, because they are formed of a various mix of *hardware*, operative systems, network technologies and programming languages.

This situation has generally historical reasons: everything that has been thought for being the most important or "the best" in a given period ended up by being added to the processing environment. For example, a few years ago Java still didn't exist, but today it would be difficult to imagine a distributed environment without at least some codes written in Java.

The prevision is that heterogeneous environments will be with us in the foreseeable future and societies are always forced to adopt *hot technologies*, such as Java, HTTP, XML or ATM.

The need to integrate applications comes from the conflict inherent heterogeneous environments. Different parts of the environment must be capable of freely communicating among themselves, even though the various technologies usually are not design to be integrated. The recent explosion of Java, *e-commerce* and Web has fully made clear that today is very difficult to be competitive in this field if you don't use software components that are completely integrated: from clients *browser* Web till *back-end* control systems.

Really, the integration of applications in an heterogeneous environment is extremely difficult. The wide variety of processing languages, network technologies, operative systems and so on, sets considerable technical problems while personalized integration solutions are not plausible from a commercial point of view, except for bigger companies, for their complexity and cost.

middleware **focus the challenge of integration supplying a common substrate of communication, not a end in itself, but integrated in a coherent way. Actually,** ***middleware*** **acts as an "integration glue" and allows developers to concentrate on the logic of the application instead of building the communication infrastructure.**

FOAM Protocol.

FOAM is a protocol, so it's simply a piece of paper on which are written some rules. In this particular case, FOAM (Free Object Access Method), is a protocol that describes how to transfer information among different applications and to call remote procedures and methods. FOAM takes inspiration by CORBA and SOAP, but it largely simplifies the job of developer, in fact it is based on standard protocols for the transport of information and on any textual format for its coding. It' useful to remember also XML (on which is based SOAP) is a textual format, so FOAM, if necessary, supports also XML, setting free the format. Basic FOAM provides for a completely asynchronous mechanism. Developer is free to introduce elements of synchronization between calling method and the called one. Thanks to FOAM and to the particular coding adopted, it's possible a multiple calling of several remote methods, set on different machines. FOAM supports, at protocol level, characteristics of **multichannelity**, *fault tolerance*[1] and *load balancing*[2], implemented in a completely new way: traditionally several servers co-operate in supplying services. FOAM follows a new philosophy in which servers compete on a virtual resource that is the supplying of the service. This approach (in line with the completely asynchronous philosophy peculiar to protocol) allows:

- a remarkable lowering of the network traffic generated by partners;
- **the functioning of the clients even when the Application Server is out** (obviuosly in this case the services required will be not supplied to the clients till one Application Server at least doesn't "go up again");
- **the access to the Internet network even in presence of a firewall**.

The idea at the base of FOAM is that communication among the parts involved in the process must be realized in a asynchronous way, in fact a synchronous exchange brings several problems:

- the systems involved must be available and functioning in a certain instant of time in which the exchange occurs;
- during the operations, generally, the systems involved have a loading peak;
- if a single action of a synchronous procedure fails, the whole procedure fails (even if it is always possible to use cunning devices so that it doesn't happen)
- the whole speed of the system is equal to the speed of the slower peripheral (because the fast peripherals must wait the answer of the slower ones).

On the contrary, integration among systems as much it is mission critical, as much it would be done in an asynchronous way. That is to say the messages exchanged by actors must be lodged in a queue manager so that to be definitively decoupled from the systems that these messages use and/or create.

Messages are the atomic element of the exchange and they are the support through which transaction is effected; they include:

- data;
- controls for the execution of remote methods;
- binary objects (for example the Adapters).
-

[1] Fault-Tolerant are said systems capable of avoiding failures even in presence of hw or sw faults;

[2] load balancing means the capacity of a distributed system of balancing the processing load among several computational agents in order to improve the whole system efficiency.

Messages (properly said FOAM DOCUMENTS) must have a set of properties (not necessarily all together):

- Lifetime: is the interval in which the document is significant (it is powerful). Out of this interval, the document is ignored;
- Identity: the univocal code which identify the document. Thanks to this property it is possible, for example, to save documents in a relational database and to obtain a series of interesting characteristics, such as the persistence of the objects, the log on the transactions, etc.;
- Acceptance: a document is "acceptable" only by the receiving nodes which set on the way to the addressee of that document or of its copies (on the base of the identity). For example: if a document contains a binary code (an adapter) for Windows, it must not be accepted by Unix/Linux machines and vice versa;
- Idempotence: the capacity of a document to be received and transmitted more than a time, having on the systems involved the same effect as if it would have been transmitted only one time;
- Receipts: each document collects the receipts of the systems it goes through. Receipts may be of "delivery" type (those collected while it is set on the way to its destination) or of "commitment" type (they certify that the document has been received by the addressee and that its content has been understood and applied).

Synchronization

Synchronization is an attribute of processes as well as of communication.

A system is said synchronous when it satisfy the following properties:
1. It exists a maximum limit of the lateness in message delivery;
2. the time needed by a process for the execution of a step is known a priori.

In a synchronous system it is possible to estimate message timeout, and this provides a mechanism for determining the failures. On the contrary, a system is said asynchronous if it doesn't exist a limit in the lateness of the message delivery, or in the time needed by a process for the execution of a step. Therefore, to say that a system is asynchronous, it must not be made any engagement on time.

The asynchronous model has obtained much more interest because it has a relatively easy semantics**; the applications developed on this model are more easily portables in comparison with applications which make engagements on time.**

In other words, in the asynchronous communication the channel logically contains a certain number of buffers of the same type, in which messages are put into order and lately sent according to a certain regulation: generally it is a FIFO queue. The execution of a send does not cause the sender waiting for the addressee having done the corresponding receive. This permits to the sender module to release, within certain limits, its functioning from that of the addressee. In some cases this may be important for avoiding deadlocks and **for increasing processing speed.**

In the synchronous communication the channels logically don't contain any buffering space. In this case the sender module and the addressee module set a sort of rendez-vous.

Synchronous modules and asynchronous modules are the two extremes of a spectrum of possible modules.

The technology: NET Services.

A program is realized in order to answer to a specific need, so that it offers one or more services. The users of a program are human beings who have the need of enjoy one or more services offered by the program.

NET Services are created with the ambition of giving an answer to the following questions:

- what does it happen when the beneficiary of a service is not a user but another program?
- how is it possible to make usable a service independently from the protocol, the platform, the Operating System and the programming language used?

The main characteristics of NET Services may be summarized as follows:

- they are accessible through a standard protocol;
- they use a textual format (eventually XML) for data exchange;
- the choice of the network technology is transparent both to the service provider and to its user (requestor);
- they permit the use of a unique technology of system integration both inside and outside a firewall;
- they support service integration independently from the platform used from the provider and so they permit the re-use of the pre-existent back-end infrastructures;
- they support multichannelity: it is necessary to free oneself from the idea that the networking context is only and exclusively the Network. NET Services are designed and realised so that to foresee several networking channels (WAP, Internet, SMS, Fax, etc.).

The architecture at the base of NET Services is based on the interaction among three main actors:

- *service provider*: the server who provides NET Service;
- *service registry*: a register which permits to find the descriptors of the service (file containing the details of the interface and of the service implementation, among which the types of data used, the operations supplied and the information needed for localising the service in the network);
- *service requestor*: the application that requires the service.

It is useful to observe that a program may act, at the same time, as service provider, service registry and service requestor, relatively or not to the same service.

The interaction among actors foresees the following activities:

- **Publication**: in order to a NET Service is accessible, it is necessary that it is published for it a descriptor which indicates to the service requestor how to interact with it;
- **Research**: through which the service requestor recovers a descriptor for the service directly or asking for it to the service registry;
- **Bind**: with Bind operation, the service requestor invokes or starts an interaction with the implementation of the NET Service lodged on the server provider.

A typical scenery of a NET Service may be the following one:

1. A service provider lodges a software module that is accessible trough a network;
2. The service provider defines a service descriptor for the NET Service and it publishes this one on a service registry or directly on a service requestor;
3. The service requestor uses an operation of research for recovering the descriptor of the NET Service and it uses this descriptor for executing the bind to the service provider and for interacting with the implementation of the NET Service.

They are expected two main modalities of interaction between the service requestor and NET Services, that are: the RMI (Remote Method Invocation) modality and the Document-Oriented modality. The first one is the classic modality of access to services supplied by traditional middleware such as EJB, CORBA, DCOM etc. The client requires a specific service and he waits for the server completing the operation. The Document-Oriented modality, on the contrary, is similar to the interaction that is obtained trough the asynchronous protocols. In this case when a client makes a request to the service, he actives a processing flow, but he doesn't wait for its conclusion.

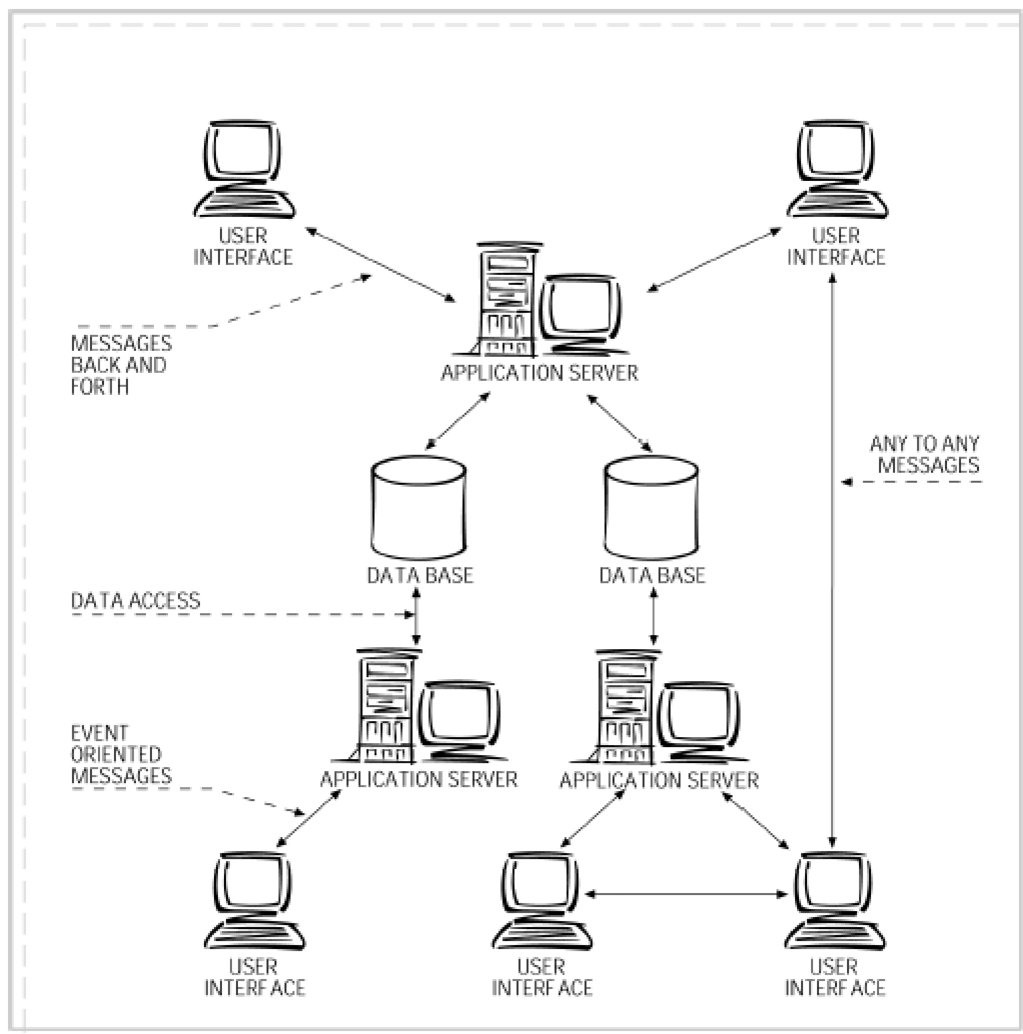

Fig.1: an example of three-thiers application where the Application Server is implemented by fault tolerance and load balancing.