Jonathan: a white paper

April 12, 2002

1 Motivations

Application programmers started developing distributed applications using plain sockets; then, they were offered higher level abstractions and tools: Remote Procedure Calls, TMs (à la Tuxedo), Message Oriented Middlewares, Object Request Brokers, and now components (Enterprise Java Beans, and soon, CORBA Components).

Each of these evolution steps resulted in a stricter separation of the business and technical aspects of distributed systems, with the following advantages:

- make the business layers easier to develop and maintain;
- specialize programmers in business aspects;
- accelerate development thanks to the reuse of technical layers developed by specialists.

However, some domains did not follow this trend, and still handle distribution and the non-functional aspects of their systems in ad-hoc, hardly re-usable ways, for different reasons:

- Runtime constraints:

  When systems have to support hard constraints (like real-time, scalability, high data rates, etc), applications may have to control precisely the system resources, or at least impose specific resource management policies.

  Most off-the-shelf products use a lot of system resources, but only offer a limited number of management policies, and usually don't let the user add new ones. In these conditions, these environments cannot be used.

- Imposed protocols

  Another limitation of the curent middleware offer is the difficulty to use communication protocols beyond the imposed standards: proprietary protocols for MOMs, IIOP for CORBA ORBs, JRMP for RMI.

  When new protocols are needed (to integrate legacy systems, to adapt a solution to a specific network, for performance reasons, etc), these products can't be used, since they don't usually let the user integrate custom protocols[1].

[1] This is also why it is difficult today to integrate MOMs and ORBs in a coherent object-oriented architecture. MOMs may be seen as special transport protocols, but nothing prevents to use a MOM to transmit inter-objects invocations. The fact is that this cannot be achieved with the currently available ORBs.

- Imposed binding models

  The most classical binding model proposed by ORB vendors is the client-server model. MOMs offer more binding models (point-to-point, broadcast), but lose the high-level object abstraction. No middleware platform offers both high-level abstractions for programming, and the possibility to manage in a flexible way the most adequate binding models for a specific application.

Jonathan is an ORB, written entirely in Java, and designed with these shortcomings in mind to provide a really adaptable middleware solution: all the internal architecture of Jonathan has been opened to allow for its adaptation to specific problems by modifying the minimum portion of code.

2 Architecture

To provide this openness and this flexibility, Jonathan has been designed on the basis of a few, yet very general and strictly applied, architectural principles. Jonathan stems for the idea that a middleware platform may be built as a composition of components, specialised to provide very specific functionalities: a buffer or thread management policy, a marshalling protocol, a communication protocol (or part of it), a security policy, a data compression component, etc. Each of these components is specialised in its own functionality, and relies on other components for all the treatments that fall out of its scope: we apply here the functional/non-functional separation principle to each micro-component of the infrastructure.

When all the functions carried out by a middleware platform have been properly separated, they have to be re-composed to yield a fully functional platform: Jonathan offers a number of assembly frameworks as a collection of APIs (Application Programming Interfaces) that must be respected by components so that they can be assembled. Four such frameworks may be found in Jonathan:

- *Binding*: this framework stricly distinguishes between the *identity* of an object and the way to access it – the *binding* –. Thanks to this framework, different binding models may be managed, or more simply, different access paths, using , e.g., different protocols, or offering different qualities of service, may be managed for the same object (i.e., a well *identified* object). In particular, this framework would enable the use of a MOM to transmit invocations between objects, in a completely transparent fashion.

- *Communication*: this framework defines the interfaces of the components implied in inter-objects communications (protocols, compression or security modules, etc), and how these elements may be composed to create bindings between objects. This framework allows for the easy introduction of new protocols in Jonathan (e.g., support for ATM or IPv6 networks, or use of a MOM as a specific transport layer.

- *Resources*: this framework defines abstractions of the management of various resources (threads, network connections, buffers), letting the programmer of resource-constrained applications introduce the most adequate resource management policies. For instance, the application programmer may choose the type of threads to be used, control their activity, use a scheduling policy based on task deadlines, etc. In the same way, it is possible to change the buffer or network connections management policies.

- *Configuration*: this framework lets the platform be configured at boot-time, and will allow its dynamical re-configuration in the future. Standard components may be replaced by new components specific to a given application, thanks to this framework.

3 Jonathan components

The main added value of Jonathan w.r.t. the other available ORBs is its flexible architecture. However, Jonathan wouldn't even exist if it didn't provide a set of components doing some real work, and enabling the development of real distributed applications !

There are today two main standards for ORBs: CORBA, from the Object Management Group, and RMI from Sun Microsystems. These two standards offer slightly different programming models, and different services and functionalities. But the Jonathan architectural principles may be applied to both standards, and a large part of the necessary components may be used whatever the chosen standard is.

Today, Jonathan supports the two main ORB standards (CORBA and RMI). The Jonathan components (corresponding to the standard functionalities of the platform) may be classified in three groups:

- common components;
- components related to CORBA;
- components related to RMI.

3.1 Common components

3.1.1 Resource management

Jonathan features components allowing to manage pools of:

- threads;
- TCP/IP connections;
- buffers.

3.1.2 Protocols

Jonathan features some basic protocols: TCP/IP, IP Multicast, and a simplified version of RTP (Real Time Protocol). Other protocols, like UDP, are in preparation[2].

3.1.3 Basic configuration tools

Jonathan offers very simple tools to configure the platform. These tools will be XML-based in a near future.

[2] IIOP (namely GIOP + TCP/IP + the CDR marshalling protocol) is of course provided, but in the CORBA components set.

3.2 CORBA components

Jonathan may be used as a CORBA ORB. It comes with the following features:

- CORBA 2.3 compliant IDL compiler;
- IIOP protocol (version 1.0)
- Dynamic Invocation Interface and Dynamic Skeleton Interface
- RMI/IIOP

Moreover, it is possible to use RTP/IP Multicast bindings, to add services and to include service contexts in IIOP messages very simply, to define “smart stubs” specific to an application, to replace the default stubs generated by the IDL compiler.

A number of basic CORBA 2.3 functionalities are still missing: in particular, the Portable Object Adapter and Object by Value support. Jonathan doesn’t offer many services, just the CORBA naming and event services. It doesn’t provide either support for security, transactions, load balancing and persistence.

The provided RMI/IIOP implementation requires a JDK 1.3 implementation to be fully standard compliant. However, it may also be used with JDK 1.2, but in this case, Java serialization is used instead of CORBA serialization. The RMI/IIOP implementation also provides some non-standard features like the possibility to avoid argument copies when the client and server are located in the same virtual machine.

3.3 RMI components

Jonathan may also be used with the RMI programming model and compilation chain. To do so, it provides a stub compiler and uses the GIOP protocol for communication.

Jonathan turns out to be much more efficient than Sun’s reference implementation, especially when the client and server are co-located in the same Java virtual machine. It also offers some non-standard features:

- a dynamic programming interface (much like CORBA’s DSI);
- support for RTP/IP Multicast bindings;
- context services transmission in the GIOP messages.

Comparing to the reference implementation, the support for server activation and distributed garbage collection are missing. A complete implementation of RMI/IIOP (with distributed garbage collection and support for the JRMP protocol) is being prepared.