

JIC: A middleware for event driven distributed objects

Matheus Gaudencio¹, Francisco Brasileiro¹

¹Laboratório de Sistemas Distribuídos (LSD)
Universidade Federal de Campina Grande (UFCG) – Campina Grande, PB – Brasil

{matheusgr, fubica}@lsd.ufcg.edu.br

***Abstract.** Nowadays distributed computing is getting more and more popular in both the academy and the industry. Still, even with more applications migrating to a distributed model of computing, frameworks and programming languages continue to offer the same old tools to work on these systems. JIC (Java Internet Communication) is a different approach to address traditional distributed computing problems with a good integration with the programming language and focus on being easy to use.*

1. Introduction

With the increase in the use of distributed systems, a greater number of tools came to assist the deployment and programming of distributed components. Technologies like RMI [Grosso 2002, RMI 2006] and CORBA [CORBA 2006] try to keep the distributed nature of the system transparent by bringing together distributed objects like in a local system. Even if this approach seems more natural to programming, is not trivial to sustain.

Distributed systems by nature has some characteristics that are intrinsic to this model of programming and that cannot be hidden without cost [Waldo et al. 1994]. First of all, partial failures may occur on those systems. In a locally deployed environment, the failure of one component will lead to the total failure of the system and, even if this does not occur, it is possible to manage those failures by one centralized entity. When the components are spread across several networks, one cannot distinguish from the communication link failure and the component failure.

Also, one must think about the concurrency scenarios that the distribute nature imposes. For instance, it is natural to any resource to receive concurrent calls from different consumers. Also, each component must know how to react when holding a lock to a shared resource; invoking a remote method while holding the lock may be a problem because the component may create a global deadlock. When mocking the local and synchronous model of a program an invoker object will block its execution thread to wait the return of the invoked method. At the remote side a new thread is created to represent this thread execution. This can lead to an unsustainable thread explosion scenario.

JIC comes as a distributed object middleware to program distributed systems and that addresses know issues of those systems. This paper presents the design of our solution and how we keep it simple to be used.

2. JIC

Event driven programming came to address the above mentioned concurrency and lock problems. With each method invocation representing an event to be processed, we can

safely hold those events in FIFO order in one event queue. It is possible to control concurrent calls with the help of one handler that translates those events to method invocations and waits for its return to execute another method. So, if objects share the same and unique event handler, they can assume that only one of them is executing on any shared memory area among them.

This can go even further. If the service using the event driven model can be safely multi-thread, it is possible to create more event handlers and by doing so, allowing more than one method to be executed in parallel, with a limited number of threads.

JIC provides a distributed object middleware that uses event driven programming (EDP) to control method invocation. But unlike other EDP frameworks as Event-Based Middleware (EBM) [Pietzuch 2002, Starovic et al. 1995] and Message-Oriented Middleware (MOM) [DS-online 2006, Monson-Haefel and Chappell 2000], it is well integrated with the programming language and works as transparent as the remote method invocation paradigm. It also provides a pooling failure detector model [Felber et al. 1999] where failures and recoveries are notified by an observer-listener model.

JIC has two main components: Access Point and Event Processor. The Access Point (AP), as depicted in Figure 1, is used to be the gateway between application and communication layers, translating method invocations into messages and vice-versa. The AP does so by holding a communication layer that works with stackable handlers: currently we use three handlers on the stack: one to control loopback invocations, other to authenticate event senders and one to control connection sessions. Also the AP has one event queue to hold incoming events, event handlers to process events and make method calls on event processors. Any object that is made remotely accessible is said to be exported as a service and the Java object to provide this service is the Event Processor (EP).

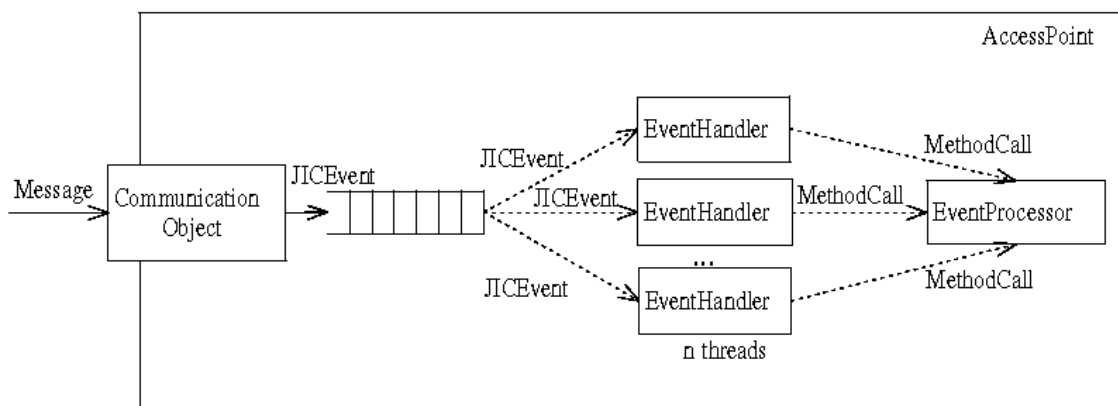


Figure 1. Access point overview

2.1. Asymmetric Connectivity

Another issue that JIC addresses is that distributed systems may pass through different administrative domains. This means that each component should know how to cope with asymmetric connectivity (i.e. presence of firewalls and NATs). Thus, any distributed object technology should provide a good communication layer to ease the deployment of entities among those domains.

We have chosen to use XMPP [JSF 2006, XMPP 2006] as the communication protocol. XMPP is a well specified protocol that is currently used by many instant messengers and applications. By defining communication servers that are public accessible and clients that can connect to those servers, any entity connected to this system can reach another entity using its XMPP address. The naming convention in this system is as follow: *user@XMPP_server/resource*.

JIC uses the AP name as resource to connect to a XMPP server. So, by knowing the remote user, XMPP server and AP name, two JIC AP can exchange messages. But since method calls in JIC are actually made on an EP providing a service, we expand this name convention to accept two more identifications, the service ID *user@XMPP_server/AP_name/service* and the object ID: *user@XMPP_server/AP_name/incarnation*, where incarnation is a number to identify which instance of one object is currently providing this service.

2.2. Semantics

JIC defines a precise set of semantics so any application using it as their communication infrastructure can understand when and why a failure is detected.

Given a scenario with two event processors (JIC objects) A and B, messages from A to B will be eventually received if both B and the channel that connects A and B don't fail. Also, those messages will be delivered in a FIFO order. If A fails and B is interested in A status, this failure will be eventually notified to B. Finally, if one message is lost from A to B, both A and B will be notified about the failure of each other.

Those semantics are guaranteed by JIC failure detection system. It is capable of detecting message loss and partial failures of entities. Moreover, it is nicely integrated with the abstraction, making explicit the need to deal with failures (just like the exception mechanism of remote method invocations).

3. Conclusion

JIC is currently being used by the *OurGrid* middleware, that is used to support the deployment of open free-to-join large scale peer-to-peer computational grid [Cirne et al.]. Initial experiments [Lima et al. 2006] shows that JIC has a performance that is comparable with RMI but provides a better thread-safe environment.

Our proposal on this paper was to explain JIC and also how JIC provides the guarantees specified, namely: i) well integration with the programming language; ii) integrated failure detection; and, iii) improved connectivity, including support for asymmetric connection. All this is provided with no perceptive performance degradation and with a very intuitive programming model.

Acknowledgments

This work has been developed in collaboration with HP Brazil R&D.

References

Cirne, W., Brasileiro, F., Andrade, N., Costa, L., Andrade, A., Novaes, R., and Mowbray, M. Labs of the world, unite!!! *Accepted for publication in Journal of Grid Computing.*

Available at walfredo.dsc.ufcg.edu.br/papers/Labs_of_the_World_Unite_v19.pdf.

CORBA (2006). *Common Object Request Broker Architecture Core Specification*. At www.omg.org/technology/documents/formal/corba_iiop.htm.

DS-online (2006). *Distributed Systems Online web site. Message Oriented Middleware section*. At dsonline.computer.org/middleware/intro.MOM.html.

Felber, P., Défago, X., Guerraoui, R., and Oser, P. (1999). Failure detectors as first class objects. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, pages 132–141, Edinburgh, Scotland.

Grosso, W. (2002). *Java RMI*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

JSF (2006). Jabber software foundation web site. At www.jabber.org/.

Lima, A., Cirne, W., Brasileiro, F., and Fireman, D. (2006). A case for event-driven distributed objects.

Monson-Haefel, R. and Chappell, D. (2000). *Java Message Service*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Pietzuch, P. R. (2002). Event-based middleware: A new paradigm for wide-area distributed systems? In *6th CaberNet Radicals Workshop*.

RMI (2006). *Java Remote Method Invocation web site*. At java.sun.com/products/jdk/rmi/.

Starovic, G., Cahill, V., and Tangney, B. (1995). An event based object model for distributed programming. In *OOIS (Object-Oriented Information Systems) '95*, pages 72–86, London. Springer-Verlag.

Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. (1994). *A Note on Distributed Computing*. Technical Report SMLI TR-94-29, Sun Microsystems Labs.

XMPP (2006). Extensible messaging and presence protocol specification. At www.xmpp.org/specs/.