Openbus 1.5.0 - Architecture Description

Maíra Gatti

August 2010

Resumo

The Openbus is a CORBA-based Enterprise Service Bus. The goal of this document is to describe the Openbus 1.5.0 release architecture. It helps on the rationale for the architecture evolution and also helps new developers to understand Openbus mechanisms details. It is not the goal of this document to present Openbus motivation or SOA concepts.

1 Arquitecture Overview

The Openbus is a component-based ESB and uses the SCS model and middleware. SCS [6] stands for Software Component System. It comprehends both a component model and a distributed service-oriented middleware, being inspired primarily by CCM and COM [8]. The Openbus goal is to provide support for component-based application integration on complex, distributed, multi-language and multi-platform scenarios. Applications that want to provide and publish their functionalities as services on the bus are granted access via certificate authentications, while clients that need to find and use such services authenticate via login and password.

Like a CCM component, a SCS component is a composition of facets and receptacles. Facets are exported interfaces, which represent services provided by the component. Similarly, receptacles are required interfaces, which represent the component's external dependencies. SCS components can then be composed together themselves to create more complex applications. This composition between components is materialized by connections between facets and receptacles that can be established at runtime. It is important to note that although facets can be called services in a SOA semantic, components can also be, because they often represent a group of tightly associated services.

The SCS model is language-independent and extensible. It specifies three basic facets available in every component, besides user-specified facets. These basic facets handle the complexities of life cycle (IComponent), connections (IReceptacles) and introspection (IMetaInterface). To allow for more adaptable components, the model doesn't impose any rules on the creation or destruction of facets and receptacles. At present, there are four implementations of the model: C++, Java, Lua and .NET/C sharp. SCS components developed in different languages interoperate through the CORBA middleware, which also provides remote communication.

The SCS middleware also provides services to support remote deployment and execution that are composed as a Deployment Infrastructure and an Execution Infrastructure. All services are components themselves and benefit from both model and middleware features. They support dynamic (re)configuration of components at runtime, including the management of deployment plans, remote installation, remote instantiation and component connections, among other features.

2 Basic Mechanisms

Before presenting the basic component-based services of which Openbus is composed, it is important to understand the basic mechanisms underline the provided basic services. There are four basic mechanisms described as following.

2.1 Leasing Mechanism

A *lease* is a contract that gives its holder specified rights over property for a limited period of time. If the lease is not renewed before expiring, the resource may become available.

Openbus uses the leasing mechanism in order to control the components access through the bus. Through the access control service, the component can get a credential associated to a lease. If the lease expires, the credential is no longer valid and the component has to get a new one.

Therefore, the client is responsible for implementing or using an existent lease renewer in order to ensure its credential will never expire, unless there is a network fault or faults like these.

2.2 Security and Authentication

In order to consume a service offer registered in Openbus, the components must provide a credential in each request. This credential is given to the client at the time that it is authenticated through the access control service.

Openbus uses CORBA Interceptors as a basic mechanism to support security and authentication. At the server side, if the client did not send its credential attached to the *service context*, the server interceptor will not be able to validate the credential and the request will be refused.

Therefore, client interceptors are responsible for automatically adding the client credential as a *service context* while the server interceptors validate this credential whenever a request is done. This security mechanism is transparent to the service implementation. Moreover, throughout all the request inside Openbus, the credential is available and any necessary further validations can be done.

2.3 Interfaces Management and Authorization

Any component that will provide a service in the Openbus is a member. The services are available through facets. Opebus has a basic mechanism for managing and authorizing those facets. This management is both done through the access control service and the registry service.

The access control service has a facet with methods for registering the members systems in Openbus. This register is based on the creation of the *System* entity followed by the addition of its deployments. The

Deployments are the *System* instances and they are the entities which really authenticates in Openbus and export interfaces (facets).

The *Deployment* identification has to be the same of the credential and the certificates must be registered by the management interface otherwise the login will not be validated by the access control service even though having the certificate into the Openbus security directory. During the Openbus installation, all the Openbus basic services are registered and it can be started without concerns.

The offers registry service has a facet which manages which services (its facets) can be exported by the *deployments*. It is possible to grant or revoke the right to export interfaces. However, only previously registered interfaces can be grants which means that there is an interface registration allowed by the registry service (the repID).

The registry service uses the *IMetaIterface* to get all the deployment interfaces at the registration time. Then it searches for the grants authorizations. It also validates the credential with the access control service.

Any information (such as to list deployments or to list authorizations) can be queried by any user, but only administrators can update them.

2.4 Fault Tolerance

In order to deliver Quality of Service (QoS), Openbus has a replication mechanism which makes it highly fault tolerant. Both the access control service and the registry service can be replicated and have a monitor. The component replica monitor runs at the same host that the component replica to be monitored. To each component replica, there is one monitor. From time to time (which can be specified by the administrator) the monitor checks if the replica is alive. The replica might not be alive because of communication failures or because it has reached an undesired state. In both cases, the monitor will:

- kill the component replica (if it is in an undesired state);
- unplug the component replica from the receptacle;
- restart another component replica;
- plug the component replica into the receptacle.

Furthermore, if several component replicas exist, they will be connected to the adaptive receptacle. If the client requests a component from this receptacle, only failure-free component replicas references will be given.

The Smart Component Proxy encapsulates the component replicas group with transparency: the client thinks that she is interacting with a single server component. It detects when a fault is raised by the requested leader server component replica and forwards the request to another one, if any available. The number of times that the Smart Component will try to forward the request while no replica is available can be defined by the administrator.

In our current solution the Smart Component Proxy contains a hash table that maps the service facets keys with their interfaces and node's address. Currently, if some of CORBA systems exceptions[1] are raised, the Smart Component will fetch another replica.

[1] *NO_RESPONSE, COMM_FAILURE, OBJECT_NOT_EXIST, TRANSIENT, TIMEOUT, NO_RESOURCES, FREE_MEM, NO_MEMORY, INTERNAL.*

The Smart Component can also be set by the administrator with regard to how many times it fetches a replica, how many time it waits for a reply, and how many time it waits between a fail reply and to fetch another replica. All those configurations can be externally updated during execution time, increasing adaptation.

Portable Interceptors do the same work that a Smart Component do: the client thinks that she is interacting with a single server component. The main difference between both is that there is a Smart Component to each dependable component. While there is only one interceptor to all dependable components and the interceptor has the reference to all the replicas in the group of each component.

ACS and Registry Service components also have specific adaptive receptacle facets specialization in order to implement replica state synchronization according to one of the presented protocols with regard to the receptacle connections synchronization.

The ACS Adaptive Receptacle implementation uses the multicast protocol to update the state if a successful connection or disconnection is done. Which means that each component replica will be invoked at the end of the request. The same behavior was implemented on the Registry Service Adaptive Receptacle.

During the Registry Service startup, it tries to connect to the ACS Adaptive Receptacle, therefore, all the active ACS component replicas will also receive this connection (if they are not already connected). In the case of the ACS startup, it will get all the connections of all active ACS component replicas and do a local connection, which means not activating the replication mechanism to avoid deadlocks.

The Registry Service component replicas access ACS Smart Component operations, rather than an ACS component replica stub and any faulty request will be transparently handled and redirected to an available ACS component replica. The ACS is not a receptacle of Registry Service because of the security design constrains set in the bus.

With regard to the compulsive leadercast protocol, all the ACS login operations (login by password, by certificate or by credential) activate the replication state using that protocol. They are the only operations which we decided to use this behavior since the credential generated by those operations is used all the time by the bus. On the other hand, we implemented the activation of this protocol as a dynamic policy: at any time the administrator can add or remove operations that activate the compulsive leadercast protocol.

On the other hand, in order to synchronize the credentials and offers between the ACS and Registry Service replicas, respectively, we use the non compulsive leadercast. Which means that unless there is a miss in any credential validation or offer's search, those data will not be synchronized.

Finally, all the clients that use the bus client library developed in Lua use Smart Components when they need a reference to the ACS component. We developed this feature through the Oil 0.5 [14] [15] facilities. While all the clients that use the bus client library developed in Java and C++ use our solution with Portable Interpcetors (those libraries use JacORB 2.3.0 [16], Orbix 6.3 sp3 [17] and MICO 2.3.13 [18]). In both cases they need to set the ACS component replicas host addresses in a configuration file and the fault manager iterative and cyclic searches for healthy replicas when a request fails. The administrator may specify how many times the fault manager is allowed to iterate over the replicas list.

3 Basic Component-based Services

[COLOCAR A FIGURA DE DIAGRAMA DE COMPONENTES SEM O SERVICO DE SESSAO]

[COLOCAR DIAGRAMA DE CLASSES SEM O SERVICO DE SESSAO]

3.1 Access Control Service Component

3.1.1 IAccessControlService Facet

3.1.2 ILeaseProvider Facet

3.1.3 IManagement Facet

3.1.4 IFaultTolerantService Facet

3.2 Access Control Service Monitor Component

3.3 Registry Service

3.3.1 IRegistryService Facet

3.3.2 IManagement Facet

3.3.3 IFaultTolerantService Facet

3.4 Registry Service Monitor Component

4 Session Service Component

[COLOCAR A FIGURA DE DIAGRAMA DE COMPONENTES COM O SERVICO DE SESSAO]

[COLOCAR DIAGRAMA DE CLASSES COM O SERVICO DE SESSAO]

4.0.1 ISessionService Facet

4.0.2 ISession Facet

5 Data Service

Referências

[1] Chappell, D. 2004 Enterprise Service Bus. O'Reilly Media, Inc.

[2] Szyperski, C. Component Software: Beyond Object-Oriented Programming. ACM Press : Addison-Wesley Publishing Co. 1998.

[3] Sun Microsystems. Enterprise JavaBeans Specification. v3.0. http://java.sun.com/ejb/

[4] OMG. CORBA Components. OMG Document formal/04-03-01 (CORBA, v3.0.3). 2004. http://www.omg.org

[5] Microsoft. Overview of the .NET Framework 4. http://msdn.microsoft.com/en-us/library/a4t23ktk.aspx

[6] The SCS Project. http://www.tecgraf.puc-rio.br/ scorrea/scs/

[7] OMG. CORBA Interceptors. OMG Document formal/04-03-01 (CORBA, v3.0.3). 2004. http://www.omg.org

[8] Microsoft COM: The Component Object Model Technologies. http://www.microsoft.com/com/

[9] Budhiraja, N., Marzulo, K., Schneider, F. B. e Toueg, S. The Primary-Backup Approach. In: Distributed Systems. Mullender, Sape (Ed.). Addison Wesley. 2nd Edition. 1993.

[10] Y. J. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M. Seri. Aqua: An adaptive architecture that provides dependable distributed objects. IEEE Transactions on Computers, 52(1):31–50, 2003.

[11] Schumacher, M. et al., Security Patterns, J. Wiley & Sons, 2005.

[12] Meling, H., Montresor, A., Helvik, B. E. and Babaoglu, O.; Jgroup/ARM: A distributed object group platform with autonomous replication management. Software: Practice and Experience, page 39,2007.

[13] Bolton, F.; *Pure* CORBA. Sams Publishing, 2002.

[14] Maia, R., Cerqueira, R. and Kon, F.; A Middleware for Experimentation on Dynamic Adaptation. In Proc. 4th Workshop on Adaptive and Reflective Middleware (ARM2005), co-located with 6th International Middleware Conference, Grenoble, France, November 2005.

[15] The Oil Project: An Object Request Broker in Lua. http://oil.luaforge.net/index.html

[16] The JacORB Project. http://www.jacorb.org/

[17] The Orbix Project. http://web.progress.com/en/orbix/index.html

[18] The Mico Project. http://www.mico.org

[19] Fraga, J., Siqueira, F., and Favarim, F. 2003. An adaptive fault-tolerant component model. In Proceedings of the 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. IEEE, Los Alamitos, CA, 179-186.