-

GigaSpaces-Mule Enterprise Service Bus for Stateful Service-Oriented Architectures

White Paper

TABLE OF CONTENTS

About this document

This document provides a short description on how to use the GigaSpaces Enterprise Edition software as an Enterprise Service Bus (ESB) runtime environment using the SymphonySoft's open source Mule ESB framework. It assumes basic knowledge and experience with the two products.
For additional information, please see –
Mule ESB Project home page: http://mule.codehaus.org/
SymphonySoft home page: www.symphonysoft.com
GigaSpaces home page: www.gigaspaces.com

Background

ESBs provide a messaging abstraction layer, enabling distributed applications to easily communicate with each other using a variety of transport mechanisms. SymphonySoft's Mule is a popular open-source product that implements the ESB model. Unlike traditional messaging approaches, the Mule ESB addresses a dynamic environment in which services can come and go, and communicate with each other on an on-demand basis and in an associative manner, based on content and using different transports depending on the application (end-point) requirements.

GigaSpaces is a leading middleware provider specializing in scaling out stateful, high-throughput, low-latency transactional applications based on its innovative Space-Based Architecture (SBA) approach. GigaSpaces enables stateful applications to easily scale up or out in a dynamic manner through the use of its underlying JavaSpaces cluster implementation, distributed caching and parallel transaction execution framework.

Putting Mule on Spaces

A tight integration between GigaSpaces and Mule provides a scalable ESB implementation with enterprise grid capabilities in a lightweight and simple to use environment. It will provide a unique value that other ESBs do not address: a solution for stateful Service-Oriented Architecture (SOA) applications.

What it means for Mule users

Integrated with GigaSpaces, Mule users can run their existing applications with no code change. The only change required is to the configuration and endpoint URLs. In this way, Mule users can leverage the scalability, robustness and performance that are a result of using GigaSpaces as the underlying runtime environment.
It also means that Mule users are able to leverage the GigaSpaces caching solution as part of their service implementation to share state between services. Combined with the GigaSpaces Service Grid Mule users would be able to **Write** their business application **Once,** and **Scale Anywhere** across the network in a transparent way, as if it were a single machine.

What it means for GigaSpaces users

GigaSpaces users can benefit from the rich functionality provided by the Mule abstraction:

- Enables POJO (Plain Old Java Object) services
- Simple integration with Web Services and other protocols
- Simple orchestration of services
- Fits nicely with the GigaSpaces Spring integration

Unique Value Proposition: ESB for Stateful SOA

In addition to the immediate need addressed by ESBs and Spaces, this section will also focus on the unique benefit that such integration provides over existing approaches. Stateful SOA applications are distributed applications that require integration with each other on two dimensions: communication and state. Many workflow scenarios fall into this category, as well as distributed computing applications. Normally, states are shared through a centralized resource – such as a file system or a database -- which often creates a scalability and performance bottleneck. Maintaining consistency in such environments also requires the use of a distributed transaction coordinator with an XA interface, which significantly impairs performance.
Typical deployment environments require three separate sub-systems: messaging, database and transaction coordination, which significantly add to the complexity of such environments from a programming perspective (complexity of APIs and number of APIs). From a deployment perspective, maintaining high availability in such systems often involves integration of separate high availability solutions for each of the three sub-systems. From a cost and licensing perspective, this approach can be relatively costly as it introduces a need for multiple licenses, often from different vendors. This lack of efficiency also results in a requirement to use additional costly hardware resources to meet the application's scalability and performance requirements.

Testimonial

Following is a testimonial from a leading architect in a financial foreign exchange institute that uses GigaSpaces and Mule:

"At our company, we're currently using Mule for post trade processing and STP.
These areas require taking in many different types of messages from different external sources using different transports, and once the message is in our system, the processing requires different business logic driven by a set of dynamic rules.
Finally, the output messages will have to go back to the source in their many different formats. We're using Mule both to leverage the many existing transport supports and because it gives us the ability to build our own transport. Also its support for rules processing such as BPEL enables us to react to ever-changing business rules.

In the future we hope to be able to combine Mule with GigaSpaces' Service-Grid (based on Rio) to build the ultimate service platform. The idea is to leverage GigaSpaces' Service Grid to provision the Mule components to the grid and use GigaSpaces' Data Grid to create a virtual memory so that Mule can transparently run in a grid environment."

GigaSpaces-Mule Solution – Space Based ESB

The GigaSpaces-Mule solution addresses the requirements described above with a simple, yet powerful approach: it provides a single integrated solution for both delivering the messages and sharing state. The solution is based on a common clustering architecture and deployment environment. The ability to share states in-memory allows applying service-oriented architectures not just as they are typically used for inter-system integration, but also as an intra-application SOA, in which the performance, latency and scalability requirements are much higher, making the classic Web Services approach irrelevant. Combined with the GigaSpaces Service Grid, the solution provides unparalleled dynamic scalability capabilities, enabling the transparent scaling out of SOA-based applications across a network of resources.

How It Works

GigaSpaces architecture in a nutshell

What is a Space?

The Tuple Space model was developed by Professor David Gelernter at Yale University in a project named Linda, back in the 1980s. The Tuple Space represents a logical shared address space via which distributed applications share information, coordinate actions (workflow), and deliver messages using a very simple set of APIs for manipulating objects in shared memory (See figure 3). During the late 90s, Sun Microsystems introduced a specification named JavaSpaces™, which is an object-oriented version of the Linda model that leverages specific capabilities of the Java language such as code mobility, reflection and embedded security.
JavaSpaces defines the API and semantics for writing and reading objects from a shared memory service. The space implementation can be virtualized. This means that it can be composed of a set of physical instances that are located in different places on the network but are still viewed and accessed as one logical instance by the application.

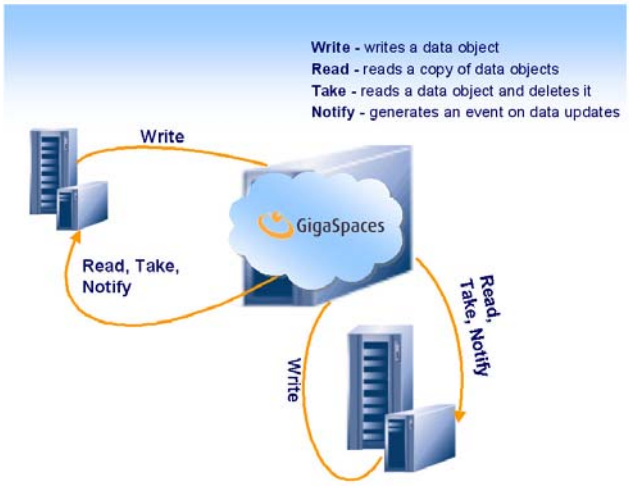


Figure 2 - The Space defines simple a API for manipulating objects in shared memory.

JavaSpaces addresses the following requirements:

Distributed Caching – providing different topologies (replicated, partitioned, embedded, persistent) for sharing data between distributed applications at in-memory speeds.

Messaging – supporting various types of messaging paradigms using a common set of APIs for all sorts of messaging:

Point-to-point
Synchronous – Through blocking *write* and *take* operations on each end – *write*, for writing a request, and *take*, for taking the return value on the sender and the opposite combination on the receiver side.
Asynchronous – Through a combination of *write*, on one end, and *take*, on the other.
Pub/Sub – Through *write*, on the sender side, and *notify* or *read*, on the receiver side.
ESB/ Content-Based Routing (CBR) -- through its query support and template matching, i.e., subscribers subscribe to messages through templates, and receive messages based on their actual attributes content.
Continuous query – Through the *notify* API, i.e., notifications are sent on any message that matches a certain query.
Workflow – Entry attributes represent states. State transitions are represented through changes to those attributes' values. Services can wait for a specific state change through a blocking *take*, with templates that represent that state. The blocking nature of the *take* operation will enable those services to wait till such an event happens.

Parallel Processing – Parallel processing refers to parallel execution of stateful transactions and not just stateless computation tasks. This is done through a combination of the master-worker pattern and partitioning. The *master* writes tasks while the *worker* takes those tasks and executes them. The *master* can write multiple tasks while a set of workers will execute those tasks in parallel. Multiple masters can execute different tasks while sharing the same pool of workers since different tasks are isolated through unique identifiers. Partitioning is used to load balance the tasks across multiple nodes, while maintaining state among the transactions. With partitioning, tasks are routed to the location of the data associated with the transaction.

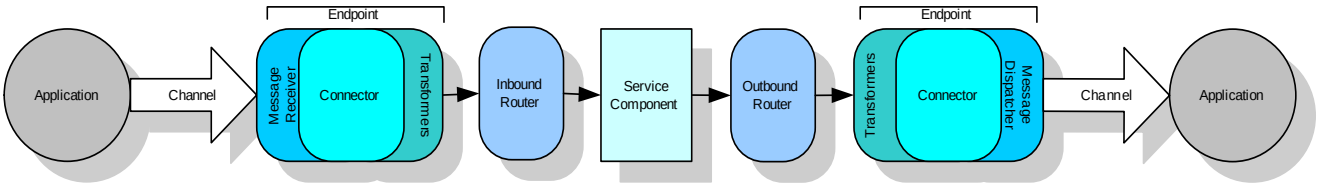
Object-Based Load-Balancing – Load-balancing is provided implicitly through the space proxy. This is different from conventional external load-balancing products that work on the protocol layer and, in most cases, by parsing HTTP headers. In an SBA, load balancing works within a virtual machine (VM) and also among VMs, and is handled implicitly without any need for additional products.

To learn more about Space-Based Messaging scenarios and how they can be applied using the space, see the following URL: http://www.gigaspaces.com/pr_dmb.html

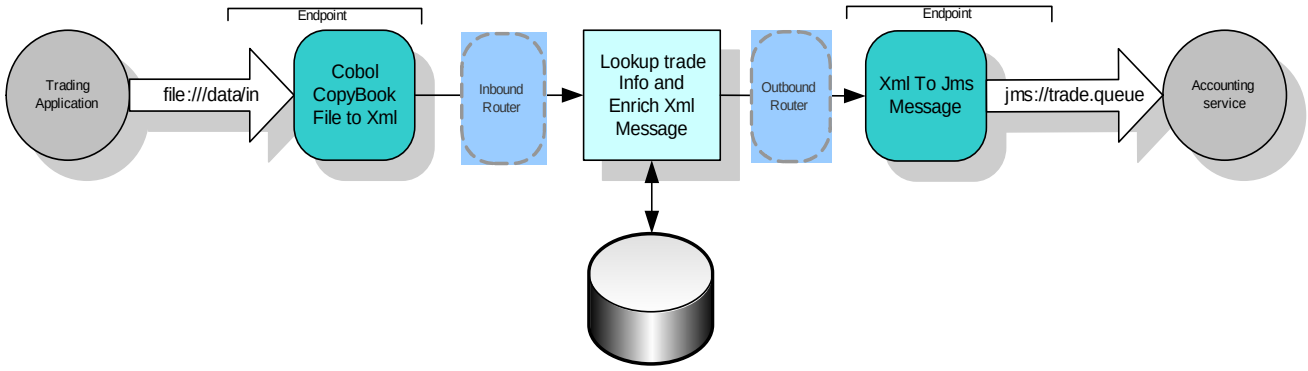
Mule architecture in a nutshell

The Mule service model is built on two basic concepts:
A Service Component. This performs some function or business logic. In Mule, a component can be any type of object such as a JavaBean, EJB Session object, and an RMI remote object, a remote web service or objects initialized in containers such a Spring, Hivemind, Plexus or PicoContainer.
An Endpoint. A communication channel for receiving and sending data. An endpoint in Mule may have data transformers associated with it to convert inbound, outbound and response data, transaction information, security information and filters.

With these two concepts, Mule allows you to 'wire' together distributed service applications that allow you to leverage your existing services and applications as well as introducing new services.

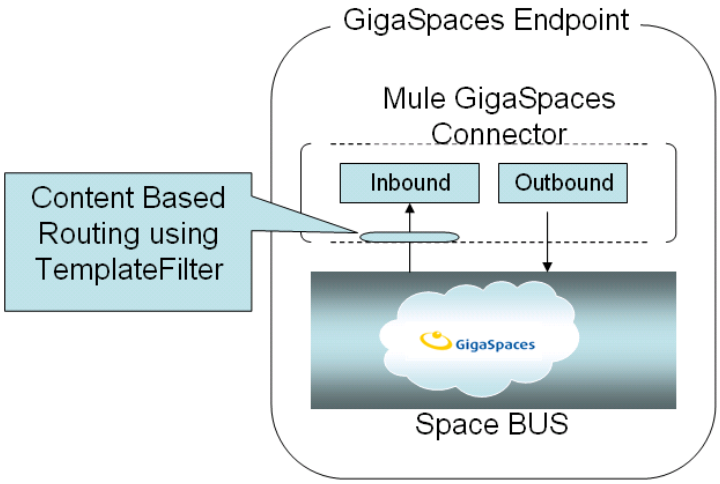


Note that the routers shown on either side of the component allow the developer to define routing logic for the way in which events are received and sent.
The following example describes a service interaction in which a file is the inbound data (the trigger for the service). The service itself is responsible for enriching the data it receives. Finally, the data is sent out on a JMS queue where it will be consumed by another application (the application could also be another Mule instance on the network).
Note the transformations that are performed on the message as it enters and exits. The file received is a Cobol CopyBook that is transformed into XML before the service receives it, and then, once the service finishes, the message will be converted into a JMS Message and sent on the 'trade.queue'.



The Mule-GigaSpaces 'Provider'

In general, the Mule-GigaSpaces provider maps inbound operations to outbound operations. The inbound adaptor takes entries and delivers them to the service implementation; the outbound filter takes an object and writes it back to the space.
The templateFilter is provided with set space templates through an XML configuration.



The GigaSpaces endpoint

The GigaSpaces endpoint provides a URL format that enables Mule services to route messages from (inbound) and to (outbound) the space. The endpoint URL format takes the following syntax:

```
<endpoint address="gs:java://localhost/Mule-space_Container/Mule-space"/>
```

The *gs:* prefix identifies which *provider* needs to be used to deliver messages for that endpoint. In our example, *gs* stands for GigaSpaces *provider*. Depending on the context in which this endpoint is defined, Mule will call the appropriate provider implementation to deliver or get messages.
The *java://localhost/Mule-space_Container/Mule-space* is the GigaSpaces URL format and is used to point to a specific space instance. In this case, the Java protocol points to a local space instance that is located in the same VM as the service implementation. The space container is named "Mule-space_Container" and the actual space instance is named "Mule-space". To point to a remote space service instance, you would use the following space URL: "*jini://*/*/Mule-space*" which will

look for an instance of a space named “Mule-space” anywhere in the network. Or "*jini://<host name>/*/Mule-space*" which will look for that space in a specific host.

For more information about the GigaSpaces URL format, see:
http://www.gigaspaces.com/docs/docs/JavaDoc/index.html?com/j_spaces/core/client/SpaceFinder.html

Receiving messages from the space using the inbound-router

To receive a message from the space, you would use the Mule inbound-router. The following declaration in the Mule service configuration will set the router to receive messages from the Order class and attribute the processed set to false.

```
<inbound-router>
<endpoint address="gs:java://localhost/Mule-space_Container/Mule-space?schema=cache">
<filter className="org.Mule.providers.gs.filters.JavaSpaceTemplateFilter"
expectedType="com.gigaspaces.esb.Order">
<properties>
        <map name="fields">
        <property name="processed" value="false"/>
        </map>
</properties>
</filter>
</endpoint>
</inbound-router>
```

Content-based routing using the template filter declaration

To define which messages need to be delivered, the space defines a template format.
The template filter enables filtering of messages based on class-name and class attributes values. Class names support inheritance relationships, which means that users can select messages of a certain class and receive not only messages of that class, but also messages that inherit from that class.
The syntax for specifying the filter using the Mule configuration is provided below:

```
<filter className="org.Mule.providers.gs.filters.JavaSpaceTemplateFilter"
expectedType="com.gigaspaces.esb.Order">
<properties>
        <map name="fields">
        <property name="processed" value="false"/>
        </map>
</properties>
```

Sending messages using the outbound-router:

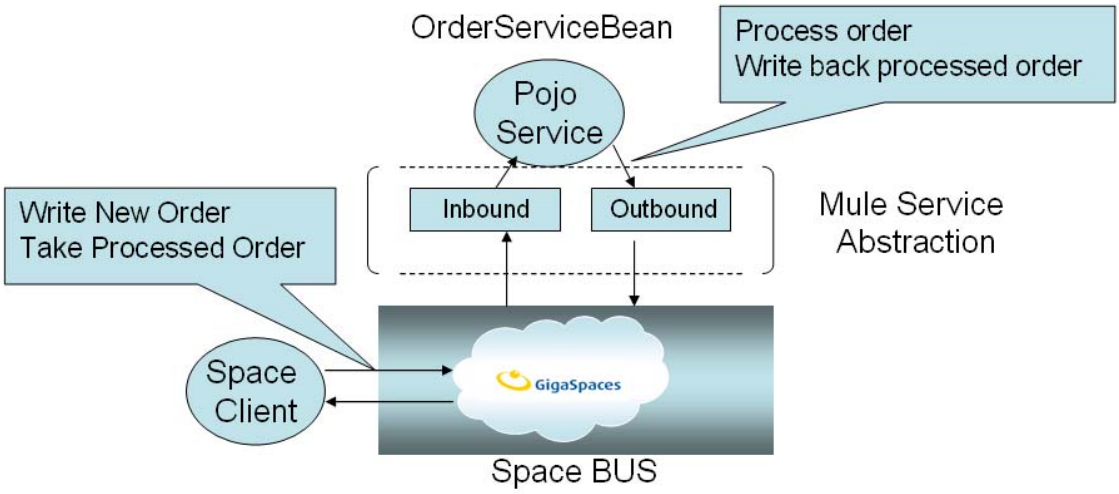
To send messages to the space we will use the following declaration in the Mule service configuration.

```
<outbound-router>
-
<router className="org.Mule.routing.outbound.ChainingRouter">
  <endpoint address="gs:java://localhost/Mule-space Container/Mule-space"/>
  <endpoint address="stream://System.out"/>

 </router>
</outbound-router>
```

Example: Order Processor Service

The order processing service is a simple service that processes Order messages and writes back to the space processed orders. The orders are submitted through a standard space client.
The following diagram describes the relationship between the different entities:



Detailed description

Step 1: Create an Order message

The Order message is a simple entry that contains two attributes:

processed – Boolean flag that indicates whether or not the order was processed.
orderId -- The Order identifier.

```
package com.gigaspaces.esb;

import net.jini.core.entry.Entry;

public class Order implements Entry {

    public String orderId;
    public Boolean processed = Boolean.FALSE;

    public Order()
    {
    }

    public Order(String id, boolean processed_flag)
    {
        orderId = id;
        processed = new Boolean(processed_flag);
    }

    public String toString(){
        return "Order{id=" + orderId + ", processed=" + processed.booleanValue() + "}";
    }
}
```

Step 2: Create The OrderServiceBean

The OrderServiceBean is a simple POJO that implements *setter* and *getter* for an Order object.
In general, the bean itself has nothing to do with either Mule or GigaSpaces. The actual "wiring" is done through the Mule configuration described below.
In general, Mule will automatically look for the *setter* method on those messages of the received message type.
In our case, this will be the Order entry.

```
package com.gigaspaces.esb;

public class OrderServiceBean {

    private Order order = null;

    // This is where the Order Processig is done
    public void setOrder(Order  order){
        System.out.println("got order " +  order);
        // Currently do nothing just mark the Order as processed
        // In reality this is where the processing logic would be executed
        _order.processed = new Boolean(true);
                this.order = _order;
    }

    public Order getOrder(){
        System.out.println("OrderServiceBean.getOrder()");
        return this.order;
    }

}
```

Step 3: Create the space client

The space client is a standard GigaSpaces client.

It will initiate the order processing chain of events by writing an Order entry to the space.

```
space.write(new Order("10",false),null, Long.MAX_VALUE);
```

It will then wait until it receives the results for the processed message using the take method:

```
Order order= (Order)space.take(new Order("10",true),null, Long.MAX_VALUE);
```

The full list is provided below:

```
package com.gigaspaces.esb;

import com.j_spaces.core.IJSpace;
import com.j_spaces.core.client.FinderException;
import com.j_spaces.core.client.SpaceFinder;

public class OrderWriter {

        /**
         * @param args
         */
        public static void main(String[] args) {
                try
```

```
                {
                        if (args.length ==0)
                        {
                                System.out.println("Usage: OrderWriter <spac url>");
                                System.exit(1);
                        }
                        System.out.println("Looking for space : " + args[0]);
                        IJSpace space = (IJSpace)SpaceFinder.find(args[0]);
                        space.write(new Order("10",false),null, Long.MAX_VALUE);
                        System.out.println("Waiting for processed order");
                        Order order= (Order)space.take(new Order("10",true),null,
                                                                Long.MAX_VALUE);
                        System.out.println("got processed order" + order);


                }
                catch(FinderException ex)
                {
                        System.err.println("failed to fine space");
                }
                catch(Exception ex)
                {
                        ex.printStackTrace();
                }

        }

}
```

Step 4: Define the Mule service configuration

The Mule service configuration is where the wiring actually happens.
In this step we will have to define a configuration that will bind the OrderServiceBean to an embedded space instance using the inbound router, and write back process events to that same space instance using the outbound router.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE Mule-configuration PUBLIC "-//SymphonySoft //DTD Mule-configuration XML
V1.0//EN"
                                "http://www.symphonysoft.com/dtds/Mule/Mule-
configuration.dtd">




<Mule-configuration id="Mule_Space_Sample" version="1.0">

    <description>
    </description>


    <!--
        The Mule model initialises and manages your UMO components
    -->
    <model name="spaceSample">
        <!--
       A Mule descriptor defines all the necessary information about how your
       components will interact with the framework, other components in the system
       and external sources. Please refer to the Configuration Guide for a full
description of all the parameters.
        -->

                <Mule-descriptor name="unprocessedOrders"
implementation="com.gigaspaces.esb.OrderServiceBean">
```

```
                        <inbound-router>
                                <endpoint address="gs:java://localhost/Mule-
space Container/Mule-space?schema=cache">
                                        <filter
className="org.Mule.providers.gs.filters.JavaSpaceTemplateFilter"
expectedType="com.gigaspaces.esb.Order" >
                                                <properties>
                                                   <map name="fields">
                                                         <property name="processed"
value="false"/>
                                                   </map>
                                                </properties>
                                        </filter>
                                </endpoint>
                        </inbound-router>

                        <outbound-router>
                                <router
className="org.Mule.routing.outbound.ChainingRouter">
                                        <endpoint address="gs:java://localhost/Mule-
space_Container/Mule-space"/>
                                        <endpoint address="stream://System.out"/>
                                </router>
                        </outbound-router>




                </Mule-descriptor>


        </model>
</Mule-configuration>
```

Step 5: Compile and run the example

Setting the environment

Install GigaSpaces from www.gigaspaces.com
Install Mule from http://mule.codehaus.org/Installation+Guide
Note – the Mule distribution contains a Community Edition of GigaSpaces as part of its distribution, but the Community Edition does not provide support for clustering. This example should be running with the Community Edition provided with the Mule distribution.
Set the MULE_HOME environment variable to point to your Mule installation.
Set the GS_HOME environment variable to point to your GigaSpaces installation.

This example assumes the following directory structure:

<example root>
bin – the directory in which you execute your commands
src – is the directory in which the source file should be located
classes – is the directory in which the example classes should be located
conf –the directory in which the Mule configuration should be located

Compiling the example

Use the following command to compile the example

Windows:

```
javac -classpath %GS_HOME%\lib\JSpaces.jar -d ..\classes
..\src\com\gigaspaces\esb\*.java
```

Unix:

```
javac -classpath ${GS_HOME}/lib/JSpaces.jar -d ../classes
../src/com/gigaspaces/esb/*.java
```

Step 6: Run the Mule service

Note that since the space is configured to run embedded within the Mule server we do need to run it separately, i.e., it's enough to run the Mule server to point it to the appropriate config file and add the space to its classpath. The command line below assumes that the ../conf/gs-Mule-config.xml file contains the configuration described in section 7.5.4 above.

Windows:

```
SET CLASSPATH=..\conf;..\classes
call %MULE_HOME%\bin\Mule.bat -config ../conf/gs-Mule-config.xml
```

Unix:

```
export CLASSPATH=../conf:../classes
${MULE_HOME}/bin/Mule -config ../conf/gs-Mule-config.xml
```

Step 7: Run the Space remote client

Windows:

```
java -classpath %GS HOME%\lib\JSpaces.jar;..\classes com.gigaspaces.esb.OrderWriter
jini://*/*/Mule-space
```

Unix

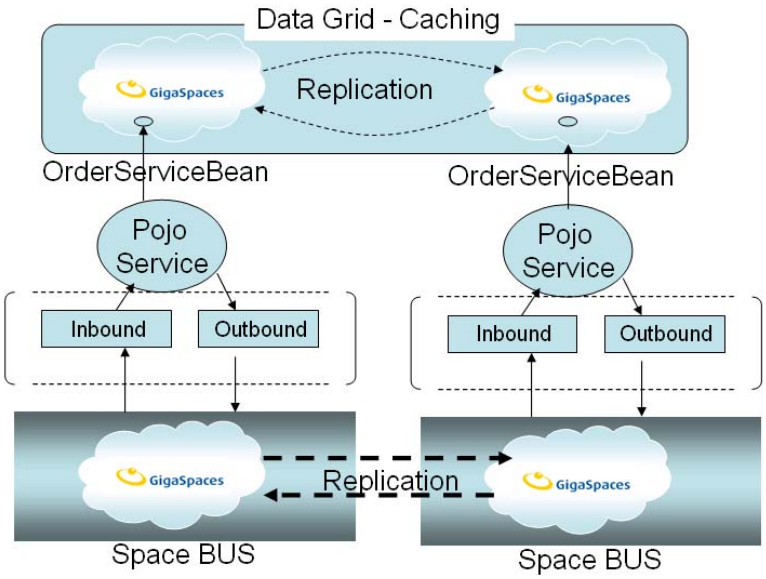
```
java -classpath ${GS_HOME}/lib/JSpaces.jar:../classes com.gigaspaces.esb.OrderWriter
jini://*/*/Mule-space
```

Common Use Cases

Stateful messaging – adding Data Grid capabilities

Stateful messaging refers to a scenario in which services share states and do not just pass events. A typical example would be a real-time analytic scenario. The request for analysis (for example, a stream of trades) flows through the bus to multiple analysis agents. All agents need to share static data (normally this would be reference data) and also share results of their calculation for computing the next stages of the process. A common solution for handling such requirement is to store the shared data in a common database. As noted earlier, such a solution suffers from multiple problems in both scalability and performance:

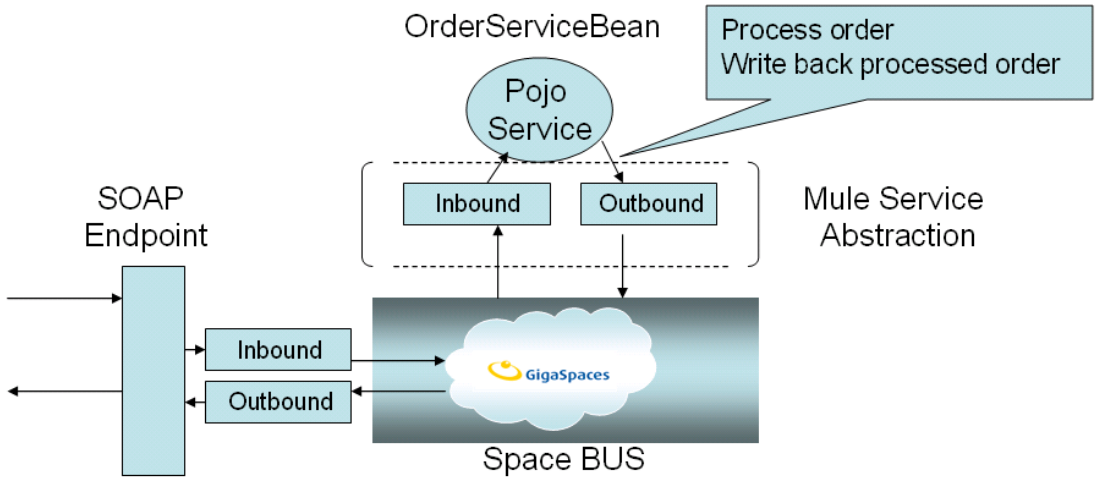
Access time to the database is relatively slow and limited by the disk I/O.
The Database is normally a shared server that introduces a scaling bottleneck
Maintaining consistency between the requests coming from the messaging bus and the database requires distributed transactions handling which add significant performance overhead.
Maintaining high-availability of the total system requires that the messaging, the transaction coordinator, and the database will all be highly available. This normally involves different clustering solutions in each layer. The result is not just reduction in the overall performance overhead, but in deployment complexity, as well as development complexity.



Putting the data in an In-Memory Data Grid (IMDG) reduces the I/O overhead and therefore allows the analytic services faster access to the shared data. The Data Grid can be either embedded within the service VM or remote in a shared memory pool. The fact that the messaging bus and the data bus use the same underlying technology and runtime environment (GigaSpaces) eliminates the complexity involved with the traditional database approach. With this option, we use common clustering deployment, a very simple API. Consistency is achieved within the space. Thus, we do not just improve the performance significantly, but we enable a new level of stability. And the best part is that it is all very simple: simple to develop, deploy and maintain.

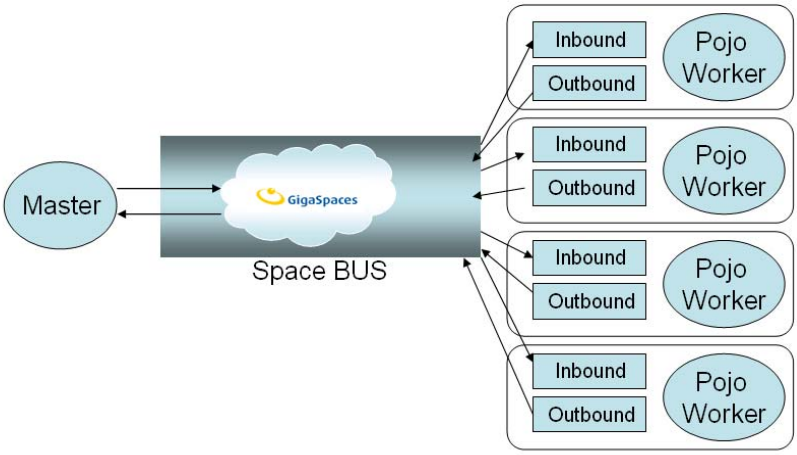
Web Services integration

One of the benefits of the Mule- GigaSpaces integration is the ability to interoperate with other transports and service frameworks.
The following example refers to a common use case for integrating with Web Services. In this example, Mule exposes the space as a web service through a web service endpoint that will route all incoming SOAP messages to the space.



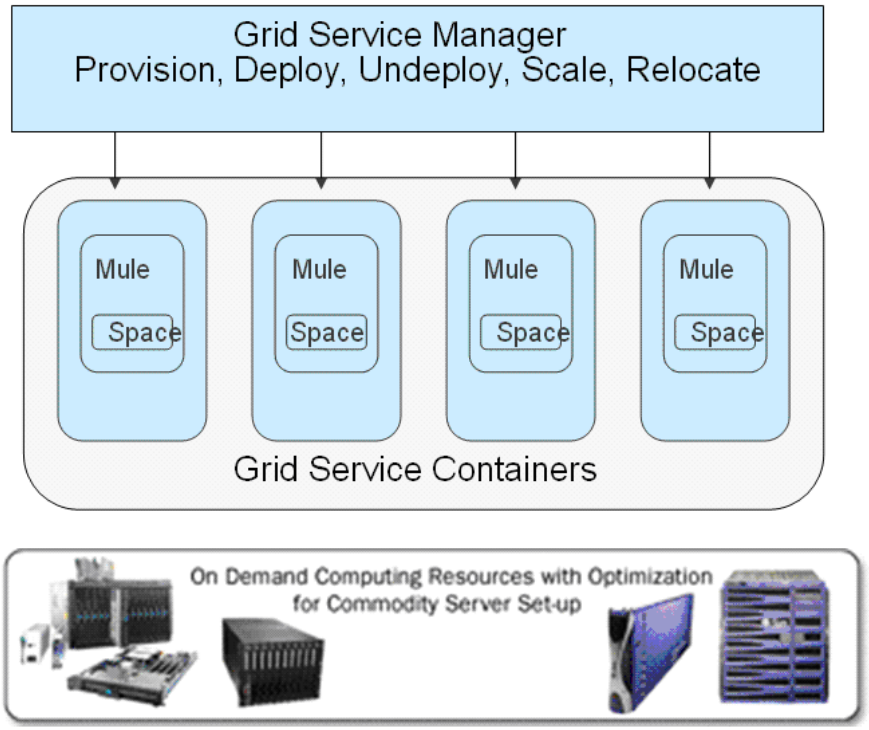
One of the inherent benefits of the above scenario is that services can be easily exposed to the web without becoming web-services themselves.

Parallel processing



The POJO workers are POJO mule services running on multiple machines and consume work that can be delivered from different transports. Workers compete on tasks delivered through the bus and process them in parallel. In this case, parallel processing does not necessarily mean number crunching (even though this could also be the case), but parallel transaction processing. In our Order example above, this could simply be Order messages that are processed in parallel on multiple machines using a set of OrderServiceBean instances.

Grid-enabled ESB using the GigaSpaces Service Grid



The GigaSpaces Service-Grid is composed of a Service Container and Grid Service Manager. The Service Container is a container that provides the ability to deploy and manage services on a remote machine. The Grid Service Manager manages the deployment and un-deployment of services, and is also responsible for re-allocation of services on alternate machines in case of a failure, or adding services in case of scaling events. Mule runs a service within the Service Grid. In this way, users will be able to dynamically deploy Mule services over a grid using a single operation, and handle failure and scaling of Mule services.

Summary - Benefits

The Mule-GigaSpcaces ESB solution turns POJOs into dynamically scalable services in a very simple way. It provides a unique value proposition over existing ESB solution enabling stateful services to share states through an IMDG (In-Memory Data Grid).

Below is a summary of the main features of the combined solution.

SymphonySoft Mule:
J2EE 1.4 Enterprise Service Bus (ESB) and Messaging broker
Large range of Transports and connectivity options
JBI Integration
Service orchestration using BPEL
Support for asynchronous, synchronous and request-response event processing over any transport.
Web Services using Axis, Glue.or XFire
Flexible deployment topologies including Client/Server, Peer-to-Peer, ESB and Enterprise Service Network
End-to-End support for routing, transport and transformation of events.
Spring framework integration. Can be used as the ESB container and Mule can be easily embedded into Spring applications.
Highly scalable enterprise server using the SEDA processing model.
Powerful event routing based on patterns in the popular EIP book.
Dynamic, declarative, content-based and rule-based routing options.
Open source community and developer mindshare.

GigaSpaces:
Common runtime environment for messaging, state sharing, workflow synchronization
Clustering and high availability for stateful applications.
Dynamically scalable runtime environment
High-performance, low-latency in-memory messaging grid.
Distributed caching.
Service Grid – enabling dynamic provisioning of the middleware as well as the business logic
Enhanced manageability
Enterprise-grade solution – proven large enterprise commercial deployments.
Global sales and marketing organization.
Vendor viability.

Try it now

You can try out the solution for FREE.

GigaSpaces provides a free Community Edition, an un-clustered version of its product.

Mule is an open-source project that contains a Community Edition of GigaSpaces as part of its distribution.

Downloadl GigaSpaces from http://www.gigaspaces.com/os_downloads.html
Download Mule from http://mule.codehaus.org/Installation+Guide

For further information or e-query contact info@gigaspaces.com