Configuring RoboServers in a BEA WebLogic Server Cluster

Kapow RoboSuite Technical Documentation

Jesper Jørgensen

Revision 2.0, November 2004
Kapow Technologies
http://www.kapowtech.com

kapow

Copyright 2003 Kapow Technologies ApS.
http://www.kapowtech.com
All rights reserved.

Contents

INTRODUCTION 5
Prerequisites 5
Overview 5
ROBOSUITE BASICS 6
RoboSuite Protocols 6
The JMS Protocol 6
The Socket Protocol 7
Client Side Distribution 7
SOLUTIONS USING THE JMS PROTOCOL 9
Simple Clustering of RoboServers 9
Advanced Clustering using JMS 9
Scenario 1: Without Distributed JMS Queues 9
Scenario 2: With a Distributed Request Queue 10
SOLUTIONS USING THE DISTRIBUTED PROTOCOL 12
Simple Clustering of RoboServers 12
Combining Client Side Distribution With Clustering 12
COMPARING THE TWO METHODS 14
Adding More RoboServers 14
RoboServer Utilization 14
Performance 14
Complexity 14
MORE INFORMATION 15
ABOUT KAPOW TECHNOLOGIES 16
APPENDIX 1: A FULL ROBOSUITE DEPLOYMENT DESCRIPTOR WITH A SOCKET PROTOCOL 17
APPENDIX 2: A FULL ROBOSUITE DEPLOYMENT DESCRIPTOR WITH A JMS PROTOCOL 18
APPENDIX 3: A FULL ROBOSUITE DEPLOYMENT DESCRIPTOR WITH A DISTRIBUTING PROTOCOL 19

Introduction

To provide increased scalability and reliability of an enterprise application running under BEA WebLogic one uses clustering. A WebLogic Server cluster consists of multiple WebLogic Server instances running simultaneously and working together. How you set up clustering in BEA WebLogic environment is not the subject of this paper but is described in **Using WebLogic Server Clusters** (Part of BEA's documentation for BEA WebLogic Server).

This paper describes how to configure applications that communicates with Kapow RoboSuite's RoboServers when these run in a WebLogic Server Environment and specifically when the WebLogic Servers are clustered. Such applications may have many RoboServers distributes on the machines of the cluster. We will describe several scenarios for how either to cluster RoboServers them selves or have RoboServers communicate with clustered WebLogic Servers.

There are essentially two ways to access Several RoboServers from client applications, e.g. a WebLogic Portal. The first is to use JMS. Using JMS you will have several RoboServers listening on the same queue and processing the request that the client or clients will put on this. For this the RoboServers will be set up to use the JMS protocol. The other way is to use *client side distribution* (available from RoboSuite 5.5). Using client side distribution the client will know which RoboServers are available and the client code (in the end the RoboSuite Java API) will use a distribution policy to divide the request between these servers. For this the RoboServers will be set up to use the Socket protocol. The code for handling Client Side distribution is part of the RoboSuite Java API and is easily deployed and configured through the API's normal configuration method: the *RoboSuite Deployment Descriptor*.

Prerequisites

We assume that the reader of this paper is familiar with RoboSuite and with how a WebLogic clustering environment works.

Overview

In this paper we will first describe the basic protocols of RoboSuite. This part is essentially just a summary of information that may also be found in the RoboSuite documentation. We will the describe solutions that use JMS and therefore also the JMS protocol to provide scalability and reliability of applications. Then we will describe the alternative that uses client side distribution and the socket protocol. Finally, we will conclude with a short discussion on the differences are between the two approaches and what their advantages and disadvantages are.

RoboSuite Basics

The section describes some RoboSuite knowledge that is relevant to the discussion later on in the paper. This is not a replacement for general knowledge of RoboSuite and assumes some basic knowledge about RoboSuite. It is compiled and included to help the reader get easy access to this.

RoboSuite Protocols

RoboServer can communicate with clients using protocols. There are two basic protocols: the Socket Protocol and the JMS protocol. In the following we will describe these and how to set them up.

There are two kinds of client applications that contacts RoboServer. The first kind is an integration application. Such applications use integration robots. These kinds of applications send input objects (defined with ModelMaker) to RoboServer and receive output objects back from RoboServer via the protocols. Though it is possible to write your own applications that communicate directly with RoboServer via the protocols, this is not advisable. Firstly, since the protocols are not documented and secondly because it would require a lot of unnecessary programming on your behalf. What you should do instead is to communicate with RoboServer using the RoboSuite Java API. This API provides all you need to communicate with RoboServer, it is documented and it has a documented mechanism for deployment: the *RoboSuite Deployment Descriptor*.

The second kind of applications that contacts RoboServer is clipping applications, e.g. in a portal application. These kinds of applications also use the RoboSuite Java API, but they do it in an indirect way. This means that you do not have to set this up yourself; it is done for you when you create your clip from RoboMaker (using the clipping wizards). The wizard will, depending on the type of portlet you are creating for you clip, create a number of files. These files may be the portlet files, a JSP files and the two configuration files: the deployment descriptor file and a clip descriptor file. The deployment descriptor file configures the way the API communicates with RoboServer e.g. what protocol to use. The clip descriptor file configures properties that specific to the clip e.g. which robot to contact and which deployment descriptor file to use. We will only need to discuss the deployment descriptor here since we are only concerned with deployment of application and not design of clips.

In the following we will introduce the protocols and talk about how to configure them though the deployment descriptor file.

The JMS Protocol

To use the JMS protocol you configure a queue, called the *request queue*, and have a RoboServer be consumers of messages from this queue. This means that you may have many RoboServer listening for requests on the same queue and you may have many client application placing requests on this queue. The JMS protocol uses three destinations: the request queue, a *response queue* and a *multicast topic*. Client sends request, e.g. for executions of a robot, to RoboServer via the request queue and receives responses to such request on the response queue. The multicast topic destination is used for administrative communication with the RoboServer, e.g. by the RoboSuite Control Center. Figure 1 shows an extract from a deployment descriptor that contains a declaration of a JMS protocol as it could look in a BEA WebLogic application. This shows the names of the three destinations `roboserver-request`, `roboserver-response`, `roboserver-multicast`, the definition of the initial context and two connection factories. To use the protocol one must define these destinations on the WebLogic server. How to do this is described in **RoboSuite RQL Service User's Guide**, a guide that is part of the standard documentation of RoboSuite.

```
<engine>
 <remote-engine>
  <jms-object-protocol
   queue-connection-factory="weblogic.jws.jms.QueueConnectionFactory"
   topic-connection-factory="weblogic.jws.jms.QueueConnectionFactory"
   request-queue-name="roboserver-request"
   response-queue-name="roboserver-response"
   multicast-topic-name="roboserver-multicast">
   <initial-context>
    <property name="java.naming.factory.initial"
     >weblogic.jndi.WLInitialContextFactory</property>
    <property name="java.naming.provider.url">t3://localhost:7001</property>
    </initial-context>
  </jms-object-protocol>
 </remote-engine>
</engine>
```

Figure 1: Configuring

Depending on the set up the request queue may be distributed or not. This is essentially transparent to RoboSuite.

If you write a simple application with one WebLogic server and one RoboServer you may have your application place request on a permanent non-distributed queue and receive the responses on another permanent queue.

You may have more than one application use the same queues since requests and responses are correlated, but this may in the end have implication on the performance of your applications and in that case you should instead use a distributed request queue. Only the request queue is distributed, the response queue is either a temporary queue generated automatically by the RoboSuite client code or a queue provide by the users application. The reason for this is that the response must reach the client that initially posted the request and not some other client and this is ensured by having the client specify the queue they want the response on. This mean that the if the client application specify a response queue in the deployment descriptor then this queue is used, but if no queue is specified in the deployment descriptor then a temporary queue is used. So the way to specify that a temporary queue should be used is by specifying an empty value for the **response-queue-name** attribute in the **jms-object-protocol** element in the deployment descriptor, i.e. like this:

```
response-queue-name=""
```

To learn more about how to configure a WebLogic Server and RoboServer to use a JMS protocol you should consult the **RoboSuite RQL Service User's Guide**.

The Socket Protocol

With the Socket protocol you configure a RoboServer to listen on a socket with a given port, default 50000. Clients then send requests and receive responses on this. Several clients may contact the same RoboServer using the same protocol. Figure 2 shows an extract from a deployment descriptor that contains a declaration of a socket protocol.

```
<engine>
  <remote-engine>
    <socket-object-protocol host="127.0.0.1" port="50000"/>
  </remote-engine>
</engine>
```

Figure 2: Configuring a socket protocol

Client Side Distribution

Beside the two kinds of basic protocols that a client may use when contacting a RoboServer there is also a distribution protocol. This protocol uses a random distribution policy to provide client side distribution and it works as follows: given a list of protocols, each time the client makes a request, one of the protocols will be chosen based on whether it is currently marked as available or not. This provides simple fail-over if at least one of the RoboServers specified in the list is available. While the distribution protocol does not explicitly provide load balancing, it can be used for that purpose.

In addition to the list of protocols the protocol has one more attribute: **retry-if-connection-lost**. This is set to true to enable support for transparent fail-over. If the connection to a RoboServer is lost while handling a request, the protocol can re-submit the request to another RoboServer from the list. In order for this to work correctly, the robot in question must be **idempotent** with respect to effect on other involved application, meaning that repeated invocations of the robot have the same effect as one. Typically this is the case with robots that do not cause permanent changes in the sites they access.

Figure 3 shows an extract from a deployment descriptor that contains a declaration of a distribution protocol. This will have three basic protocols in its list and have the **retry-if-connection-lost** set to true. We have only shown socket protocols in this example, but the distribution protocol also works with JMS protocols or a mixture of basic protocols.

```
<engine>
  <distributing-engine retry-if-connection-lost="true">
    <random-distribution-policy>
      <remote-engine>
        <socket-object-protocol host="localhost" port="50000"/>
      </remote-engine>
      <remote-engine>
        <socket-object-protocol host="10.10.10.85" port="50000"/>
      </remote-engine>
      <remote-engine>
        <socket-object-protocol host="10.10.10.86" port="50001"/>
      </remote-engine>
    </random-distribution-policy>
  </distributing-engine>
</engine>
```

Figure 3: Configuring a distributed protocol

The only distribution policy currently available is a random distribution policy. The just distribute request randomly between the available RoboServer with equal weight on all servers. It is possible to get a simple form of weighed distribution by adding the same server several times to the list of available servers. If one adds a server twice then this will in average be called twice as often as a server that occurs only once.

Solutions Using the JMS Protocol

In this section we will describe solution that uses the JMS Protocol.

Simple Clustering of RoboServers

A simple form of clustering of RoboServers may be used in the case where no clustering of the client application is needed, but the load on the RoboServers is the bottleneck. For this one only have to configure the WebLogic Server as JMS Provider for one set of destination and then start as many RoboServers as needed to obtain the required response time. That is there is no need for a distributed request queue. These RoboServers should be configured to use the JMS protocol with the queues on the WebLogic Server. The client application contacts the RoboServer through the RoboSuite Java API (RSJA), which in turn use the JMS protocol to contact the Servers. The Java API makes sure that request and responses are correlated, so that responses for several concurrent requests do not get mixed up (in a portal application using clipping this happen automatically). The RoboServers have no state so for application, e.g. clipping, where there has to be some correlation between several consecutive requests it does not matter which of the RoboServers handles each of the request. Figure 4 shows the setup for this kind of clustering.

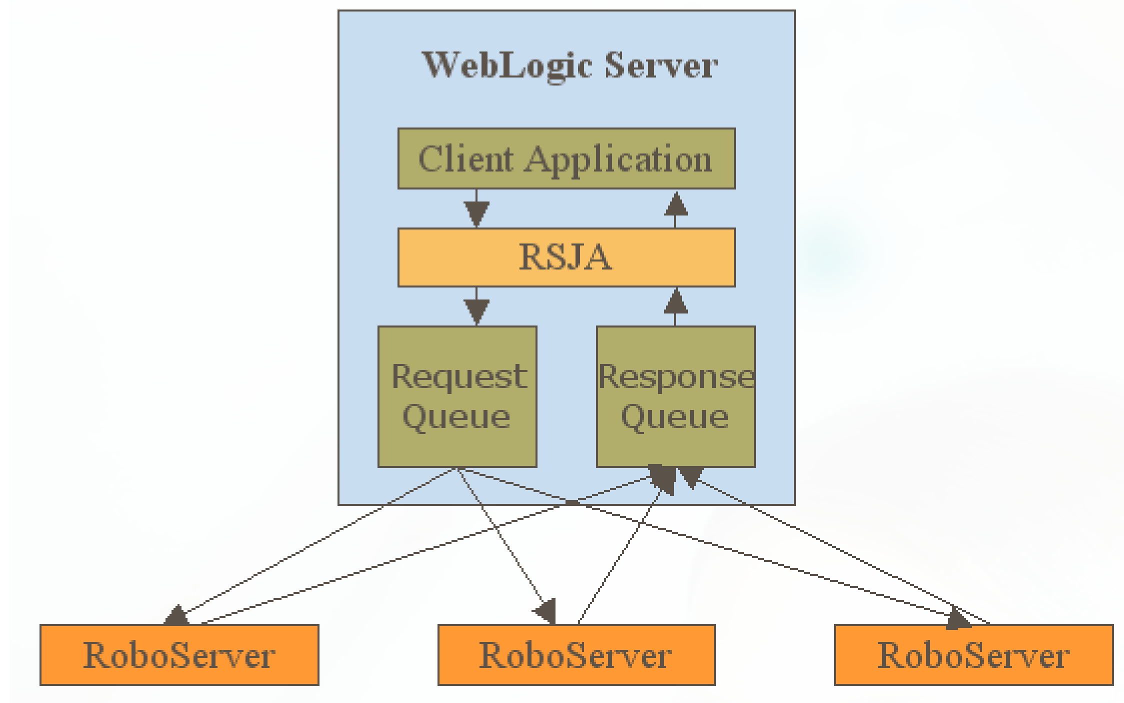

Figure 4: Simple Clustering of RoboServers using JMS

If more RoboServers are need in the cluster these can be added dynamically, by simply starting more RoboServers using the same JMS protocol settings.

Advanced Clustering using JMS

The simple clustering example describe in the previous section is not using a distributed queue and in most cases this is all you need. In some cases the client application is running on a WebLogic Server cluster and in that case one need to consider how to set up RoboServers to communicate with the WebLogic Servers in the cluster.

Scenario 1: Without Distributed JMS Queues

The simplest way to have application running in a WebLogic cluster contact RoboServers is to use the setup from the previous section on every server in the cluster using identical non-distributed queues on each server. This setup is shown in Figure 5. In this the client application on each WebLogic Server will place request messages on their own request queue and will receive response on their own response queue.

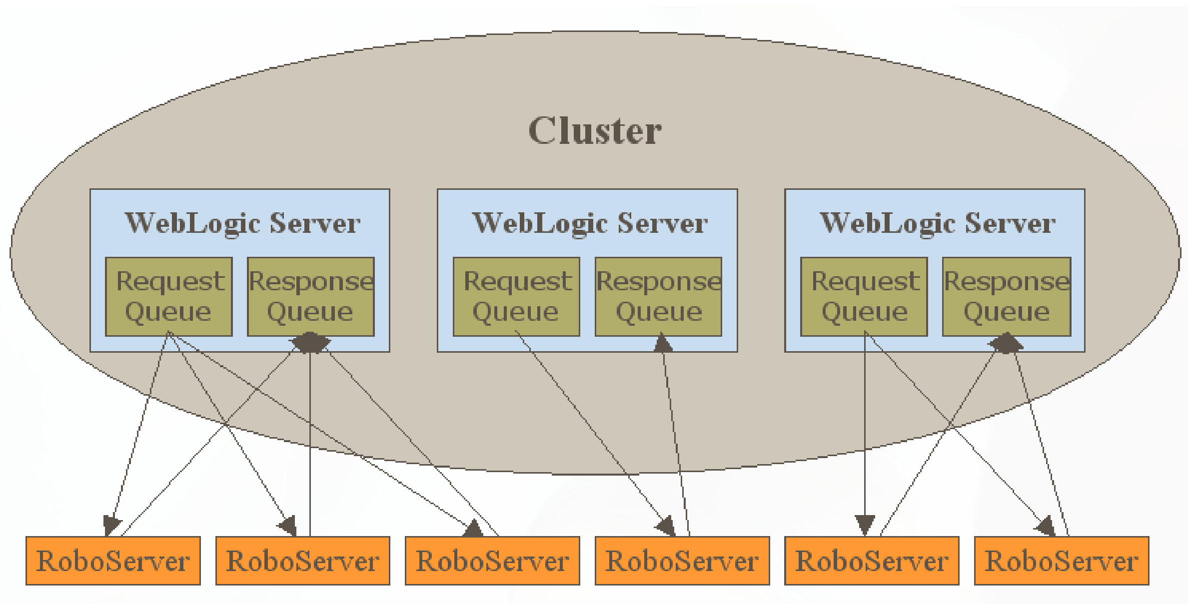

Figure 5: A Cluster without a Distributed Request Queue

The number of RoboServers attached to each WebLogic server in the cluster should be dimensioned in such a way that the RoboServers are able to ensure the performance of the single Server they are attached to is acceptable. This mean the if a client application, e.g. a Portal, on one server receives three times are many request as a client application on an other server, then the first server need to have enough RoboServers to be able to handle three times as many RoboServer requests as the other server. For example, if WebLogic Server 2 in Figure 5 needs exactly one RoboServer to for the client application to meet its performance demands and WebLogic Server 1 receives three times as many requests as WebLogic Server 2, then WebLogic Server 1 need three RoboServers. Since one is the minimal number of RoboServer that a WebLogic Server need (if it want to run robots) there might be cases where a RoboServer is not running at 100% of its capacity. Let us, as another example assumes that the WebLogic Server 2 is only using its RoboServer 50%. In that case WebLogic Server 1 only needs two RoboServers, because this is enough to ensure that these are not overloaded. Even though there is no load balancing for the RoboServers the application might still have load balancing, since the client application (the portal application) may still have load balancing.

Scenario 2: With a Distributed Request Queue

In this scenario the client application in the cluster will contact RoboServer using a distributed request queue and a non-distributed response queue. This setup may give an even better load balance than Scenario 1 since an overloaded WebLogic Server may be able to distribute its robot request to all and not just its own RoboServer.

Distributed JMS destinations are used in WebLogic for clustering JMS. This provides reliability and scalability for JMS resources. Of these two benefits the most important is reliability. If the JMS service is not clustered, it will be a single point of failure in the communication with the RoboServers. Scalability of the JMS service is not likely to be a concern, unless there is a very large number of RoboServers. Typically, several RoboServers can process requests from a single JMS queue.

A distributed destination represents one or more distributed destination members. A destination member is always pinned to a particular server member of a cluster. Accessing the distributed destination is done almost transparently, but it does require some additional effort to provide failover of the JMS resource. The document **Programming WebLogic JMS** (Part of BEA's documentation for BEA WebLogic Server) provides more detail on distributed JMS destinations.

Requests and responses are sent on separate queues. Only the request queue should be distributed. If the response queue is distributed, responses will be distributed among several queue members. This makes it impractical for the client to pick up the responses, and we loose ordering guarantees on the received responses. Therefore, the setup of clustered JMS for RoboServer communication requires a mix of distributed and non-distributed destinations. Figure 6 shows the setup. In this the request queue (Req. Q) is a distributed queue. Client on any of the WebLogic Servers place request on this and the request are then distributed among the instance queues according to the load-balancing heuristics for distributed queue. Server 2 may also have its own member queue. The response queues are not shown in the figure since it is up to the client application how this implemented; it could be either permanent queues or temporary queues. Figure 6 show this. The messages are not physically placed on the distributed queue (Dist. Req. Q)

as the figure might suggest. The arrows are only indicating the fact that from the client applications aspect the messages are places in this queue. The arrow going from the distributed queue to the real request queues indicates that the clustered JMS implementation will eventually place the messages on these queues for the RoboServers to pick up.

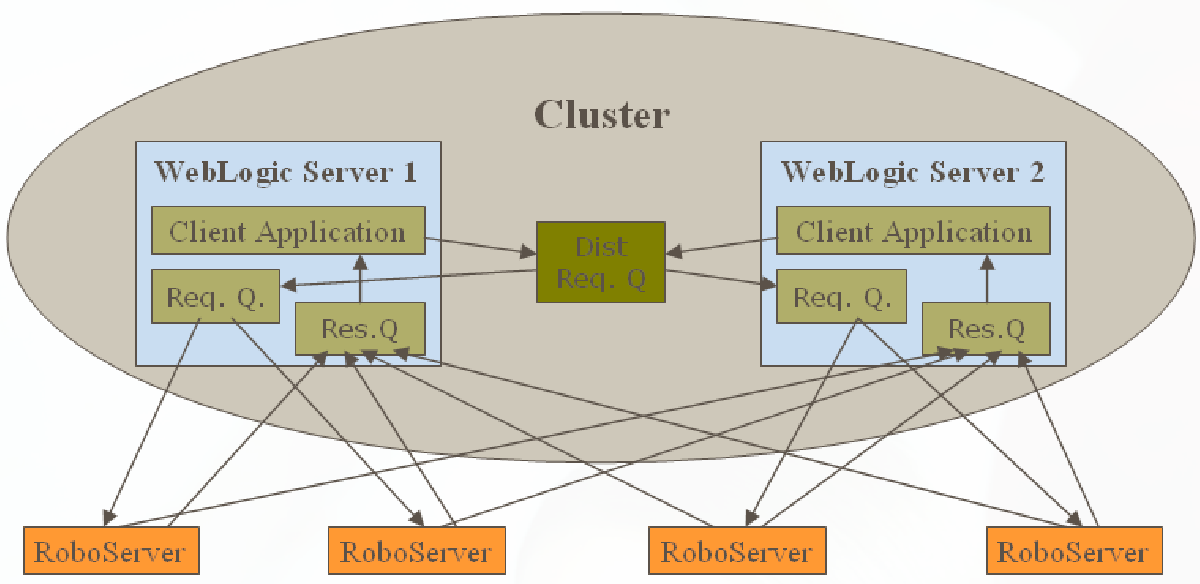

Figure 6: A Cluster with a Distributed Request Queue

Solutions Using the Distributed Protocol

In this section we will describe solution that uses the Socket Protocol.

Simple Clustering of RoboServers

A simple form of clustering of RoboServers may be used in the case where no clustering of the client application is needed, e.g. a Portal application, but the load on the RoboServers is the bottleneck. To do this with the Socket protocol you need to configure the client such that it uses a Client Side Distributing Engine. Such an engine is part of the RoboSuite Java API (RSJA). This can be configured to distribute request to many RoboServers. In a clipping solution this configuration is part of the clipping wizard, but may also be done manually by changing the RoboSuite Deployment Descriptor of the project. Figure 7 shows the set-up for this kind of clustering.

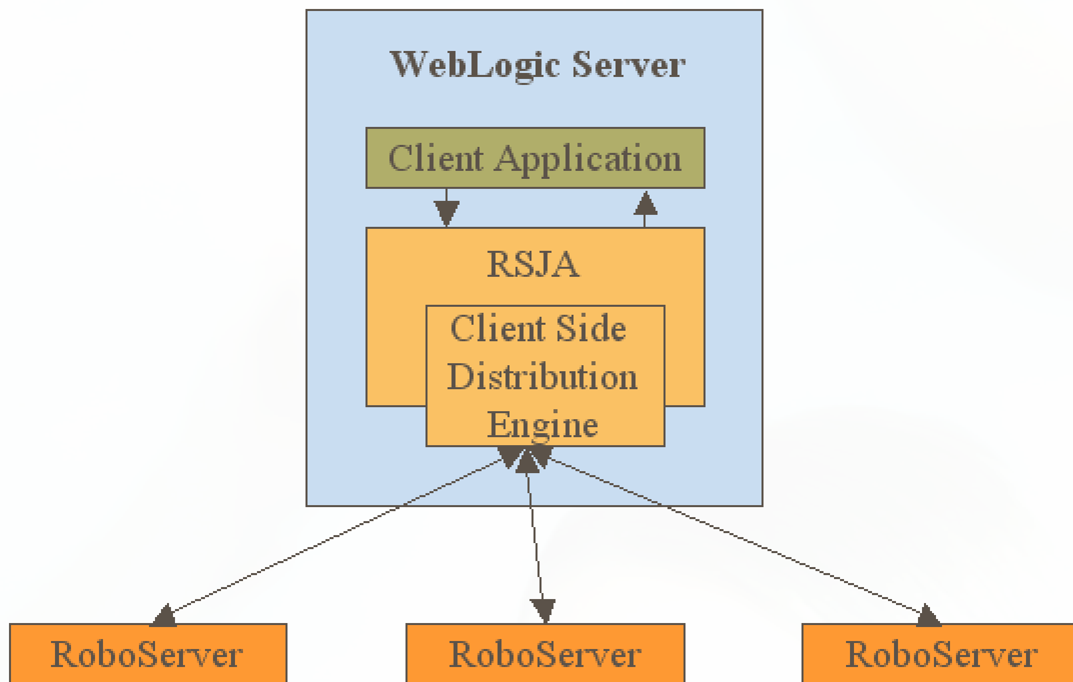

Figure 7: Simple Clustering of RoboServers

Combining Client Side Distribution with Clustering

In this final example we will combine the clustering of the client application e.g. a Portal application with client side distribution of RoboServers. In this the application is running on a cluster of WebLogic servers. Each of these the client application is configured to call the RoboServers using a Client Side Distributing Engine. Essentially each server is configured as the single server in the Simple Client Side Distribution example, but the different servers may or may not share RoboServers. Figure 8 shows the set-up for this kind of clustering. In many cases it is probably both the best and also the simplest solution to have all the servers use the same list of RoboServers, since this will probably ensure the best utilization of the RoboServers and that no RoboServers are ever idle as long as there is still one WebLogic server running.

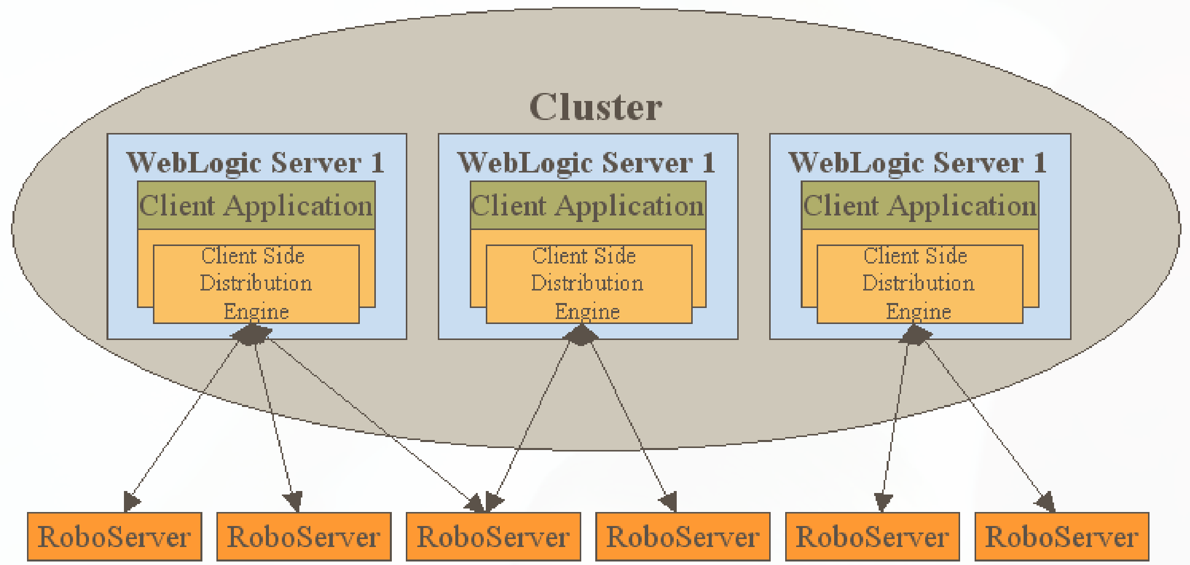

Figure 8: A Cluster with Client Side Distribution

Comparing the Two Methods

In this section we will give a brief comparison of the two methods discussed: Distributed JMS and client side distribution.

Adding More RoboServers

For JMS it is easy to add a new RoboServer since one does just set up a new RoboServer to listen for request on a give queue. There is no need to change the configuration of the application to do this. For Client Side distribution adding a new server is a bit more complex since this would mean that one would have to change the application and redeploy this. Essentially one has to change an XML configuration file and up date this on all servers.

RoboServer Utilization

If a WebLogic server is down for some reason, e.g. hardware failure, maintenance, etc. will any of the RoboServers become idle? For non-distributed JMS a RoboServer only performs request for one WebLogic Server. This means that each WebLogic server will be sure to have a given number of RoboServers handling requests for it without any other servers interfering, but it also means that if a WebLogic server is down then all its RoboServers will be idle.

For distributed JMS this is not the case since all RoboServers get their request from the same distributed queue.

For Client Side distribution a RoboServer may perform request for more than one WebLogic Server. This mean that if all RoboServers in a Cluster has been set up to receive requests from at least two WebLogic Servers then if one WebLogic server is down then there will not necessarily be any RoboServers idle.

Performance

Client side distribution will most often use the socket protocol and experience has shown that the socket protocol is about 50% faster than the JMS protocol which used by the distributed JMS solution. This is at least the case if one use the default set up for JMS. For JMS one has to consider how persistence is managed. If messages are persisted in a database or in a file system in stead of in memory, this may have a considerably degrading effect on performance. You should consult an expert on JMS on WebLogic to help you set this up.

Complexity

The JMS protocol is considerably more difficult to set up than the socket protocol. So for demos, PoC's, development and even for smaller production system it might be better to use the socket protocol. One can then for production systems consider whether the use of JMS is required to obtain the required reliability and load balancing.

More Information

Kapow Technologies web site:
http://www.kapowtech.com

Kapow Developer Connection web site:
http://kdc.kapowtech.com

RoboSuite BEA WebLogic Edition:
http://bea.com/framework.jsp?CNT=index.htm&FP=/content/products/kapow

Patent applications:
PCT/DK00/00163, PCT/DK00/00429, PCT/DK00/00700

About Kapow Technologies

Profile:

Founded in June 1998, Kapow Technologies is a world-leading supplier of software for integration to web-enabled applications.

Kapow Technologies offers a wide variety of options to customers, ranging from web clipping and data collection to advanced application integration, all using the unique RoboSuite platform.

Selected partners:

BEA, IBM, Software AG, ATG, Autonomy

Selected customers:

Lycos, NATO, Danske Bank, NPD, APR Smartlogik, The Arlington Institute, TDC, BetBrain, The Danish National IT and Telecom Agency, Krak.

Contact information:

Kapow Technologies
Dr. Neergaards Vej 5A
DK-2970 Hørsholm
Denmark

Tel +45 70 33 10 00
Fax +45 70 33 10 01

http://www.kapowtech.com

Appendix 1: A Full RoboSuite Deployment Descriptor with a Socket Protocol

Figure 9 show a deployment descriptor for a client that uses the socket protocol to contact RoboServer. The RoboServer is located on the same machine as the client and is listening on port 50000. The descriptor also specifies that the robot library used is going to be sent to the server embedded in the request.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE deploy PUBLIC
  "-//Kapow Technologies//DTD RoboSuite Deployment Descriptor 1.2//EN"
  "http://www.kapowtech.com/robosuite/rql/dtd/deployment-descriptor_1_2.dtd">
<deploy>
  <engine>
    <remote-engine>
      <socket-object-protocol host="127.0.0.1" port="50000"/>
    </remote-engine>
  </engine>
  <robot-library>
    <embedded-file-robot-library>/embedded.robotlib
    </embedded-file-robot-library>
  </robot-library>
</deploy>
```

Figure 9: A Deployment Descriptor with a Socket Protocol

Appendix 2: A Full RoboSuite Deployment Descriptor with a JMS Protocol

Figure 10 shows a deployment descriptor for a client that uses the JMS protocol to contact RoboServer.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE deploy PUBLIC
   "-//Kapow Technologies//DTD RoboSuite Deployment Descriptor 1.2//EN"
   "http://www.kapowtech.com/robosuite/rql/dtd/deployment-descriptor_1_2.dtd">
<deploy>
  <engine>
    <remote-engine>
      <jms-object-protocol
        queue-connection-factory="weblogic.jws.jms.QueueConnectionFactory"
        topic-connection-factory="weblogic.jws.jms.QueueConnectionFactory"
        request-queue-name="roboserver-request"
        response-queue-name="roboserver-response"
        multicast-topic-name="roboserver-multicast">
        <initial-context>
          <property name="java.naming.factory.initial">
            weblogic.jndi.WLInitialContextFactory
          </property>
          <property name="java.naming.provider.url">
            t3://localhost:7001
          </property>
        </initial-context>
      </jms-object-protocol>
    </remote-engine>
  </engine>
  <robot-library>
    <embedded-file-robot-library>/embedded.robotlib
    </embedded-file-robot-library>
  </robot-library>
</deploy>
```

Figure 10: A Deployment Descriptor with a JMS Protocol

Appendix 3: A Full RoboSuite Deployment Descriptor with a Distributing Protocol

Figure 11 shows a deployment descriptor for a client that uses a distributing protocol (client-side distribution) to contact RoboServer. The client will distribute request between three server where the two are located on the same machine as the client (localhost and 127.0.0.1) and the other on a machine with the IP address: 192.168.6.72.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE deploy PUBLIC "-//Kapow Technologies//DTD RoboSuite Deployment
Descriptor 1.2//EN" "http://www.kapowtech.com/robosuite/rql/dtd/deployment-
descriptor_1_2.dtd">
<deploy>
  <engine>
    <distributing-engine retry-if-connection-lost="true">
      <random-distribution-policy>
        <remote-engine>
          <socket-object-protocol host="localhost" port="50000"/>
        </remote-engine>
        <remote-engine>
          <socket-object-protocol host="127.0.0.1" port="50000"/>
        </remote-engine>
        <remote-engine>
          <socket-object-protocol host="192.168.6.72" port="50000"/>
        </remote-engine>
      </random-distribution-policy>
    </distributing-engine>
  </engine>
  <robot-library>
    <embedded-file-robot-library>/embedded.robotlib
    </embedded-file-robot-library>
  </robot-library>
</deploy>
```

Figure 11: A Deployment Descriptor with a Distributing Protocol