Clustering

Cluster capabilities are provided by *JBoss Clustering* using a *Managed Domain*. See the References section for links that describe both these features of JBoss. Using these technologies, bridge instances can be deployed on multiple physical *host* and combined into groups to form multiple managed bridge groups.



While this scenario is possible, it is much simpler to create a single bridge instance on each host and have a single bridge group that is managed by the Domain Controller.



One thing to keep in mind is that the while the diagram shows the Domain Controller running on its own host, it can easily run on either Host 1 or Host 2. There is no need to have the Domain Controller running in its own host.

## Master/Slave relationship

The JBoss documentation uses the term *master* and *slave* domain/host controller but is a bit weak in the explanation of what these are. Essentially, these terms refer more the setup of the host.xml configuration file. Domain controllers are simply Host Controller with special responsibilities. They take their configuration from the host.xml file but Domain Controllers use certain portions of the host.xml file that have special meaning. Specifically, if the domain-controller element (see below) contains <local/>, then that host is designated as the *master* domain controller and all other host controller are *slave* domain/host controllers.

## Configuration

In order to configure a domain, you will need to configure the following file:

<JBOSS\_HOME>/domain/configuration/host.xml

<JBOSS\_HOME>/domain/configuration/domain.xml

The first thing to do is to decide which host will be the domain controller and configure it to act as a domain controller. This is done by modifying the host.xml file. Edit this file and locate the domain-controller element. Make sure this element contains the <local/> sub-element:

<domain-controller>

   <local/>

</domain-controller>

This tells JBoss that the localhost will operate as a domain controller for the cluster.

Next you must expose a native (i.e. non-HTTP) management interface on an address that is accessable to all hosts in the domain. In the host.xml file, make sure it contains this:

<management-interfaces>

  <native-interface security-realm="ManagementRealm">

    <socket interface="management" port="${jboss.management.native.port:9999}"/>

  </native-interface>

  <http-interface security-realm="ManagementRealm">

    <socket interface="management" port="${jboss.management.http.port:9990}"/>

  </http-interface>

</management-interfaces>

The interface attributes above refer to a named interface declaration that you can find later in the host.xml file. This interface declaration will be used to resolve a corresponding network interface used to contact the domain controller:

<interfaces>

   <interface name="management">

       <inet-address value="[ip-address]"/>

   </interface>

</interfaces>

The “[ip-address]” should be the IP address of the host acting as the domain controller. Note that you must us the IP address of the physical interface. You cannot use “localhost” nor “127.0.0.1”.

You must then create an “admin” user ***on the domain controller.*** This user will be used by other, slave host controllers to connect. To do this you must run the “add-user” script located in the <JBOSS\_HOME>/bin directory. See the References section for details on running this utility.

At this point the domain controller should be configured and you can now configure host controllers that will join the cluster. Start by modifying the host.xml file and change the name attribute in the host element to be a unique name:

<host xmlns="urn:jboss:domain:1.0" name="BridgeHost1">

[...]

</host>

Next we need to configure a security realm to be used to manage all bridge instances. Modify the host.xml file on all slave controllers and add the following security realm:

<security-realm name="SlaveBridgeManagementRealm">

  <server-identities>

    <secret value="cE3EBEkE=" />

  </server-identities>

</security-realm>

## ProtocolStack Object

A *ProtocolStack* is an object that implements a communications protocol used by the bridge to send and receive data to and from remote devices. Protocol stacks consist of a number of *ProtocolLayer* objects that pass data between each other. Each ProtocolLayer communicates with the layer directly above and below it.

Below the ProtocolStack is a *TransportAdapter* that is responsible for communication data to and from a network source. Currently, the only transport adapter available is TCP.

Above the ProtocolStack is the Application that handle application specific logic.



The bridge code uses two interfaces to implement ProtocolLayers: UpperLayer and LowerLayer. Each defines methods used by adjacent layers to send and receive data:

**public** **interface** UpperLayer {

**public** **void** setLowerLayer(LowerLayer lowerLayer);

**public** **void** handleReceivedData(ByteBuffer byteBuffer) **throws** ProtocolException;

}

**public** **interface** LowerLayer {

**public** **void** setUpperLayer(UpperLayer upperLayer);

**public** **void** transmitData(ByteBuffer byteBuffer) **throws** IOException;

**public** **void** close();

}

For convenience these are combined in to a single interface called *ProtocolLayer*

The setUpperLayer and setLowerLayer are used during the build process of a ProtocolStack to wire up the layers of the stack. A ProtocolLayer layer passes data to its upper layers via the UpperLayer’s handleReceivedData method and to its lower layer the LowerLayer’s transmitData method. LowerLayer also defines a close() method that should be used to close the layers. Because layers can be dependent on each other, when a LowerLayer’s close() method is called, the layer should take action to clean up and then invoke the close() method on its adjacent lower layer. Since the ProtocolStack maintains the layers that make up the stack, it will ensure that that the close method is invoked on the topmost layer in the stack.

### Threading

The TransportAdapter and ProtocolStack are controlled by a single thread within the bridge’s network subsystem. Each layer that makes up the protocol stack should not create its own thread to access other layers in the stack. Doing so might require locking other parts of the stack, which could lead to deadlock conditions. For this reason, ProtocolLayers need not worry about being thread safe (assuming all the rules are followed) but they ***must be re-entrant***, meaning that if a call is made to an upper or lower layer, it’s possible that the same thread may invoke a callback method on the same layer.

For example, consider a stack made up of a framing layer below an acknowledgement layer. In a case like this, it’s possible that that framing layer will receive a datagram, decodes it and passes the decoded data up to the acknowledgement layer but before returning, the same thread could generate an acknowledgement packet that is passed it to the framing layer for transmission.

If a layer needs to perform some kind of asynchronous function, it can request an *Asynchronous Trap* (AST) from the ProtocolStack’s owning thread. An AST is requested invoking requestTrap method an AstRequester object and passing an AstHandler object. When an AST is requested, the stack’s owner thread will invoke the AstHandler.handleAsynTrap() method and allow a ProtocolLayer or a ProtocolStack to perform an operation within the context of the owning thread. AstHandler code should complete as quickly as possible and avoid taking out any locks. Since the AstHandler runs within the context of the ProtoclStack’s owning thread, it can access upper and lower layers.

## Building a ProtocolStack

A ProtocolStack is specified through the bridge’s configuration file, pacbridge.xml, using the <Protocol> element in the network definition section. The simplest method is to create a <Protocol> element and specify name attribute. Within the protocol element, you can specify optional properties to be set on the protocol. As an example, the following shows the protocol definition for STP:

<Protocol name=*"STP"*>

<MaxPacketSize>2048</MaxPacketSize>

<TransmitTimeoutSeconds>120</TransmitTimeoutSeconds>

<ReceiveTimeoutSeconds>120</ReceiveTimeoutSeconds>

</Protocol>

In order for the bridge to use this protocol, class must be created contains a @Protocol annotation with a name parameter of STP. E.g.:@Protocol(name = "STP")

**public** **class** StpProtocolStack **implements** ProtocolStack {

**public** StpProtocolStack() {

...  
 }

**public** **void** setTransmitTimeoutSeconds(Integer transmitTimeoutSeconds) {

...  
 }

**public** **void** setReceiveTimeoutSeconds(Integer receiveTimeoutSeconds) {

...  
 }

**public** **void** setMaxPacketSize(Integer packetSize) {

...  
 }

}

The name argument of the @Protocol annotation is case insensitive. At startup time, when the bridge encounters the <Protocol name=”...”> element, it will search for a class annotated with the @Protocol annotation that matches the name specified in the element. Any class that contains this annotation must be public, provide a no-argument constructor, and must implement that ProtocolStack interface. Note that only classes in the “zedi.pacbridge” package and sub-packages are searched.

Any properties defined in the <Protocol> element will be set using reflection, which means that the protocol stack class must provide java beans style setter methods for these properties.

If a ProtocolStack implementation needs to perform asynchronous activities, either by itself or by some of its layers, it can request that an AstRequester object be injected by annotating a method the with @AsyncRequester annotation. The name of the method is arbitrary but it must accept a single AstRequester object:

@AsyncRequester

**public** **void** giveMeMyAstRequester(AstRequester requester) {

**this**.astRequester = requester;

}

## DefaultProtocolStack

The bridge provides a single threaded implementation of ProtocolStack. This class can be used to implement a protocol defined by an array of ProtocolLayer objects or can be used to derive a different implementation.

When constructing a DefautlProtocolStack, an array of ProtocolLayers is passed to the constructor.

**public class DefaultProtocolStack** **implements** ProtocolStack {

. . .

**public** DefaultProtocolStack(ProtocolLayer[] protocolLayers) {

. . .  
 }  
}

The array of layers is wired up from the first element to the last, meaning that ProtocolLayer[0] will be the top most layer and ProtocolLayer[n-1] will be the lower most layer of the protocol stack.

If DefaultProtocolStack is used to derive and a new protocol stack, a no-argument protected constructor is provided for derived classes. Within its constructor, the derived class should construct an array of ProtocolLayer object and call the protected “setProtocolLayers” method.

The DefaultProtocolStack provides and setAstRequester annotated method to have an AstRequester object injected. If the derived class requires an AstRequester, it can override the base implementation but it must call the super class’s method.

Here is the complete code for the StpProtocolStack:

@Protocol(name = "STP")

**public** **class** StpProtocolStack **extends** DefaultProtocolStack **implements** ProtocolStack {

**private** Fad fad = **new** Fad();

**private** Apl apl = **new** Apl();

**public** StpProtocolStack() {

**this**.fad = **new** Fad();

**this**.apl = **new** Apl();

ProtocolLayer[] layers = **new** ProtocolLayer[]{fad, apl};

setProtocolLayers(layers);

}

**public** **void** setTransmitTimeoutSeconds(Integer transmitTimeoutSeconds) {

fad.setTransmitTimeoutSeconds(transmitTimeoutSeconds);

}

**public** **void** setReceiveTimeoutSeconds(Integer receiveTimeoutSeconds) {

fad.setReceiveTimeoutSeconds(receiveTimeoutSeconds);

}

**public** **void** setMaxPacketSize(Integer packetSize) {

fad.setMaxPacketSize(packetSize);

apl.setMaxPacketSize(packetSize\*2);

}

@Override

@AsyncRequester

**public** **void** setAstRequester(AstRequester requester) {

**super**.setAstRequester(requester);

fad.setAstRequester(requester);

}

}

## Event Processing Coordinator

The bridge receives outgoing control requests via *events* published to a queue on a Message Server. However, because remote devices can only communicate with a single bridge instance, only one bridge in the cluster can process the control. To avoid multiple bridges from processing outgoing controls, only a single bridge in the cluster is reads events from the event queue and processes them. This bridge is known as the EventProcessingCoordinator. Which bridge becomes the *Event Processing Coordinator* is determine at bridge startup.

Each bridge will fork a thread that will compete for a cluster wide lock called *pacbridge$event\_queue\_processor*. If the thread acquires the lock, it will register an object to process events from the event queue and that bridge will then become the Event Processing Coordinator. If the thread is unable to acquire the lock, it will block until the lock is acquired. If the EventProcessingCoordinator bridge crashes, or is shutdown, it will release the *pacbridge$event\_queue\_processor*. There will be a brief transition period where another node in the cluster will acquire the lock and become the Event Processing Coordinator.

Events that come off the event queue are outgoing control requests.

# Deploying the WSMQ Resource Adapter

Follow the instructions on [this](https://access.redhat.com/site/documentation/en-US/JBoss_Enterprise_Application_Platform/6/html/Administration_and_Configuration_Guide/Deploy_the_WebSphere_MQ_Resource_Adapter.html) page. For 7.5.0.0 you don’t need to worry about mqetclient.jar.

References:

JBoss Cluster and Managed Domain:

<https://docs.jboss.org/author/display/AS71/Core+management+concepts>

Domain Setup

<https://docs.jboss.org/author/display/AS71/Domain+Setup>

Adding a user with the add-user utility

<https://docs.jboss.org/author/display/AS71/add-user+utility>

File locations:

Permanent

Design/Concepts

\\esfile\Research\DHF\MKT-298 LCLP FI System Platform