Multiplexer (MUX)

Table of Contents

Diameter Multiplexer (MUX) Design 1
Diameter Multiplexer (MUX) Setup 1
Pre-Install Requirements and Prerequisites 2
Source Code 2
Diameter Multiplexer (MUX) Configuration 3
Diameter MUX Source Overview 3
Diameter Multiplexer (MUX) Dictionary 5

The Multiplexer (MUX) is designed as a stack wrapper. It serves two purposes:

Expose Stack

It exposes the stack and allows it to be shared between multiple listeners. The stack follows the life cycle of the MUX. It is created and destroyed with MUX.

Expose Management Operations

Exposes the management operations for clients, one of them being the &MANAGEMENT.PLATFORM; Console. For specific information please refer to the Restcomm Diameter Management Console User Guide.

Diameter Multiplexer (MUX) Design

MUX is a simple service provided on behalf of the Stack. Entities interested in receiving messages for certain diameter applications register in MUX. Upon registration, the entity passes a set of application IDs. Based on the message content and registered listeners, MUX either drops the message or passes it to the proper listener. MUX checks application IDs present in the message to match the target listener.

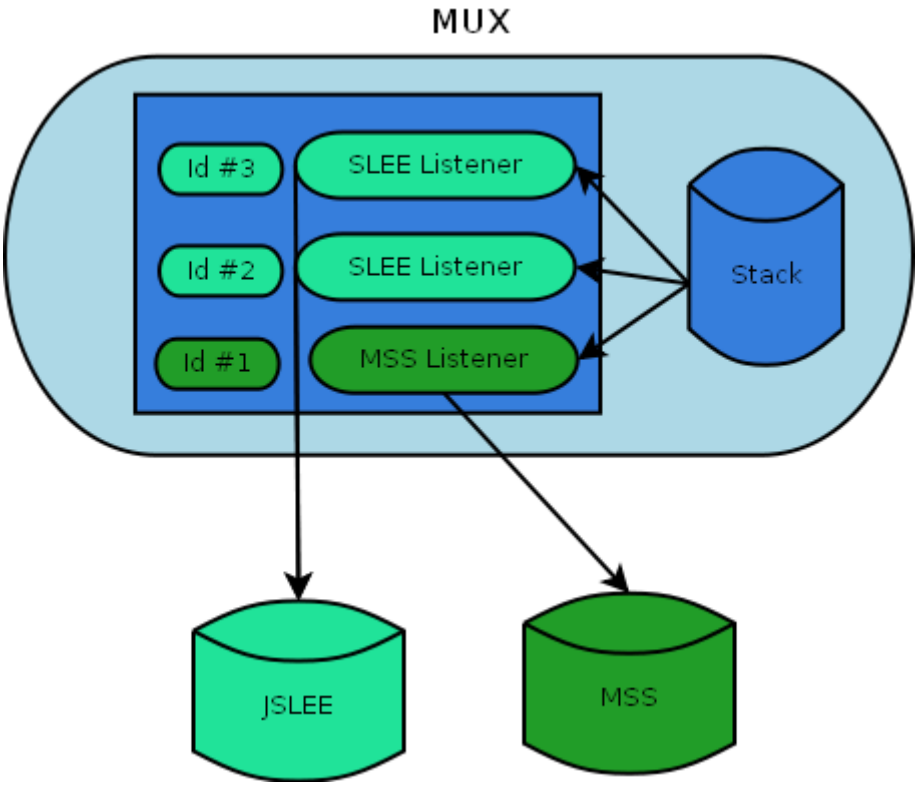

Figure 1. Diameter Multiplexer (MUX) Design Overview

Diameter Multiplexer (MUX) Setup

Pre-Install Requirements and Prerequisites

Ensure that the following requirements have been met before continuing with the installation process.

Hardware Requirements

MUX does not have any hardware requirements.

Software Prerequisites

MUX must be deployed either in {jee-version} v4.x or v5.x. However it is possible to adapt configuration files and run in any JMX container.

Source Code

This section provides instructions on how to obtain and build the Restcomm Diameter MUX from source code.

Release Source Code Building

1. Downloading the source code

Subversion is used to manage its source code. Instructions for using Subversion, including installation, can be found at http://svnbook.red-bean.com.

Use SVN to checkout a specific release source. The base URL is https://github.com/Restcomm/jdiameter . Then add the specific release version, for example 1.7.0-SNAPSHOT .

```
[usr]$ git clone git@github.com:RestComm/jdiameter.git
```

2. Building the source code

Maven 3.2.5 (or higher) is used to build the release. Instructions for using Maven2, including installation, can be found at http://maven.apache.org.

Use Maven to build the deployable unit binary.

```
                    [usr]$ cd -
                    [usr]$ mvn install
```

Once the process finishes you should have the SAR built. If the `JBOSS_HOME` environment variable is set, the will be deployed in the container after execution.

By default Restcomm Diameter MUX; deploys in the JBoss Application Server v5.x . To change it, run `maven` with the profile switch command: `-Pjboss4`.

Development Trunk Source Building

Follow the Release Source Code Building procedure, replacing the SVN source code URL with https://github.com/Restcomm/jdiameter .

Diameter Multiplexer (MUX) Configuration

MUX requires three configuration files:

jboss-service.xml

This file is specific to SAR. Please refer to the ` manual for explanation. The file binds Diameter MUX under the following JMX object name by default: `diameter.mobicents:service=DiameterStackMultiplexer`.

jdiameter-config.xml

This file configures the stack exposed by MUX. Please refer to [_jdiameter_configuration] for details. It is located in *mobicents-diameter-mux-.sar/config*.

dictionary.xml

This file configures the dictionary. Its structure and content is identical to the file described in [_jdiameter_validator_configuration].

Diameter MUX Source Overview

The Diameter MUX capabilities are defined by the `MBean` interface `org.mobicents.diameter.stack.DiameterStackMultiplexerMBean`. This interface defines two types of methods:

Management

Used by RHQ console. These methods are outside the scope of this documentation.

Stack Accessors

Methods that allow the user to retrieve and use a wrapped stack.

The methods below are of interest to a Diameter MUX user:

```
public Stack getStack();
```

Returns a stack wrapped by the multiplexer. It is present as a convenience method, and the stack should only be changed directly by expert users.

```
public void registerListener(DiameterListener listener, ApplicationId[] appIds) throws
IllegalStateException;
```

Registers a listener to be triggered when a message for a certain application ID is received.

`public void unregisterListener(DiameterListener listener);`

Removes the message listener.

`public DiameterStackMultiplexerMBean getMultiplexerMBean();`

Returns the actual instance of MUX.

The listener interface is defined below:

```
package org.mobicents.diameter.stack;

import java.io.Serializable;

import org.jdiameter.api.Answer;
import org.jdiameter.api.EventListener;
import org.jdiameter.api.NetworkReqListener;
import org.jdiameter.api.Request;

public interface DiameterListener extends NetworkReqListener, Serializable,
    EventListener<Request, Answer>
{

}
```

MUX can be used as follows:

```
public class DiameterActor implements DiameterListener
{
    private ObjectName diameterMultiplexerObjectName = null;
    private DiameterStackMultiplexerMBean diameterMux = null;

    private synchronized void initStack() throws Exception {
        this.diameterMultiplexerObjectName =
            new ObjectName("diameter.mobicents:service=DiameterStackMultiplexer");

        Object[] params = new Object[]{};
        String[] signature = new String[]{};

        String operation = "getMultiplexerMBean";
        this.diameterMux=mbeanServer.invoke(this.diameterMultiplexerObjectName,
operation,
            params, signature);

        long acctAppIds = new long[]{19312L};
        long acctVendorIds = new long[]{193L};
        long authAppIds = new long[]{4L};
        long authVendorIds = new long[]{0L};
        List<ApplicationId> appIds = new ArrayList<ApplicationId>();
        for(int index = 0;index<acctAppIds.length;index++) {
            appIds.add(ApplicationId.createByAccAppId(acctVendorIds[index],
acctAppIds[index]));
        }

        for(int index = 0;index<authAppIds.length;index++) {
            appIds.add(ApplicationId.createByAuthAppId(authVendorIds[index],
authAppIds[index]));
        }

        this.diameterMux.registerListener(this, appIds.toArray(new ApplicationId
[appIds.size()]));
        this.stack = this.diameterMux.getStack();
        this.messageTimeout = stack.getMetaData().getConfiguration().getLongValue(
            MessageTimeOut.ordinal(), (Long) MessageTimeOut.defValue());
    }
}
```

Diameter Multiplexer (MUX) Dictionary

The Dictionary is part of the MUX package. Its purpose is to provide unified access to information regarding AVP structure, content and definition. It is configured with an XML file: *dictionary.xml*.

Dictionary logic is contained in the `org.mobicents.diameter.dictionary.AvpDictionary` class. It exposes the following methods:

`public AvpRepresentation getAvp(int code)`

Return an `AvpRepresentation` object representing the AVP with the given code (assuming vendor ID as 0 (zero)). If there is no AVP defined, it returns `null`.

`public AvpRepresentation getAvp(int code, long vendorId)`

Returns an `AvpRepresentation` object representing the AVP with the given code and vendor ID. If there is no AVP defined, it returns `null`.

`public AvpRepresentation getAvp(String avpName)`

Returns an `AvpRepresentation` object representing the AVP with the given name. If there is no AVP defined, it returns `null`.

Dictionary uses a POJO class to provide access to stored information: `org.mobicents.diameter.dictionary.AvpRepresentation`. It exposes the following methods:

`public int getCode()`

Returns the code assigned to the represented AVP.

`public long getVendorId()`

Returns the vendor ID assigned to the represented AVP.

`public String getName()`

Returns name assigned to the represented AVP. If no name is defined, it returns `null`.

`public boolean isGrouped()`

Returns `true` if the AVP is of grouped type.

`public String getType()`

Returns a `String` with the name of the represented AVP type. Return value is equal to one of defined types. For example, `OctetString` or `Unsiged32`.

`public boolean isMayEncrypt()`

Returns `true` if the AVP can be encrypted.

`public boolean isProtected()`

Returns `true` if the AVP *must* be encrypted. This occurs if `public String getRuleProtected()` returns `must`.

`public boolean isMandatory()`

Returns `true` if the AVP must be supported by an agent to properly consume the message. It only returns `true` if `public String getRuleMandatory()` returns `must`.

`public String getRuleMandatory()`

Returns the mandatory rule value. It can return one of the following values: `may`, `must` or `mustnot`.

`public String getRuleProtected()`

Returns the protected rule value. It can have one of the following values: `may`, `must` or `mustnot`.

`public String getRuleVendorBit()`

Returns the vendor rule value. It can have one of the following values: `must` or `mustnot`.

The Diameter MUX Dictionary can be used as follows:

```
public static void addAvp(Message msg, int avpCode, long vendorId, AvpSet set, Object
avp) {
    AvpRepresentation avpRep = AvpDictionary.INSTANCE.getAvp(avpCode, vendorId);

    if(avpRep != null) {
        DiameterAvpType avpType = DiameterAvpType.fromString(avpRep.getType());

        boolean isMandatoryAvp = avpRep.isMandatory();
        boolean isProtectedAvp = avpRep.isProtected();

        if(avp instanceof byte[]) {
            setAvpAsRaw(msg, avpCode, vendorId, set, isMandatoryAvp, isProtectedAvp,
(byte[]) avp);
        }
        else
        {
            switch (avpType.getType()) {
            case DiameterAvpType._ADDRESS:
            case DiameterAvpType._DIAMETER_IDENTITY:
            case DiameterAvpType._DIAMETER_URI:
            case DiameterAvpType._IP_FILTER_RULE:
            case DiameterAvpType._OCTET_STRING:
            case DiameterAvpType._QOS_FILTER_RULE:
                setAvpAsOctetString(msg, avpCode, vendorId, set, isMandatoryAvp,
isProtectedAvp,
                    avp.toString());
                break;

            case DiameterAvpType._ENUMERATED:
            case DiameterAvpType._INTEGER_32:
                setAvpAsInteger32(msg, avpCode, vendorId, set, isMandatoryAvp,
isProtectedAvp,
                    (Integer) avp);
                break;

            case DiameterAvpType._FLOAT_32:
                setAvpAsFloat32(msg, avpCode, vendorId, set, isMandatoryAvp,
isProtectedAvp,
                    (Float) avp);
                break;

            case DiameterAvpType._FLOAT_64:
                setAvpAsFloat64(msg, avpCode, vendorId, set, isMandatoryAvp,
isProtectedAvp,
                    (Float) avp);
                break;

            case DiameterAvpType._GROUPED:
                setAvpAsGrouped(msg, avpCode, vendorId, set, isMandatoryAvp,
isProtectedAvp,
```

```
                    (DiameterAvp[]) avp);
                break;

            case DiameterAvpType._INTEGER_64:
                setAvpAsInteger64(msg, avpCode, vendorId, set, isMandatoryAvp,
isProtectedAvp,
                    (Long) avp);
                break;

            case DiameterAvpType._TIME:
                setAvpAsTime(msg, avpCode, vendorId, set, isMandatoryAvp,
isProtectedAvp,
                    (Date) avp);
                break;

            case DiameterAvpType._UNSIGNED_32:
                setAvpAsUnsigned32(msg, avpCode, vendorId, set, isMandatoryAvp,
isProtectedAvp,
                    (Long) avp);
                break;

            case DiameterAvpType._UNSIGNED_64:
                setAvpAsUnsigned64(msg, avpCode, vendorId, set, isMandatoryAvp,
isProtectedAvp,
                    (Long) avp);
                break;

            case DiameterAvpType._UTF8_STRING:
                setAvpAsUTF8String(msg, avpCode, vendorId, set, isMandatoryAvp,
isProtectedAvp,
                    (String) avp);
                break;
            }
        }
    }
}
```