Diameter Stack Source overview

Table of Contents

Diameter stack is built with the following basic components:

Session Factory

The Session Factory governs the creation of sessions - raw and specific application sessions.

Raw and Application Sessions

Sessions govern stateful message routing between peers. Specific application sessions consume different type of messages and act differently based on the data present.

Stack

The Stack governs all necessary components, which are used to establish connection and communicate with remote peers.

For more detailed information, please refer to the Javadoc or the simple examples that can be found here: Git Testsuite HEAD.

Session Factory

`SessionFactory` provides the stack user with access to session objects. It manages registered application session factories in order to allow for the creation of specific application sessions. A Session Factory instance can be obtained from the stack using the `getSessionFactory()` method. The base `SessionFactory` interface is defined below:

```
package org.jdiameter.api;

import org.jdiameter.api.app.AppSession;

public interface SessionFactory {

    RawSession getNewRawSession() throws InternalException;

    Session getNewSession() throws InternalException;

    Session getNewSession(String sessionId) throws InternalException;

    <T extends AppSession> T getNewAppSession(ApplicationId applicationId,
        Class<? extends AppSession> userSession) throws InternalException;

    <T extends AppSession> T getNewAppSession(String sessionId, ApplicationId
        applicationId, Class<? extends AppSession> userSession) throws
InternalException;
}
```

However, since the stack is extensible, it is safe to cast the `SessionFactory` object to this interface:

```
package org.jdiameter.client.api;


public interface ISessionFactory extends SessionFactory {

    <T extends AppSession> T getNewAppSession(String sessionId,
        ApplicationId applicationId, java.lang.Class<? extends AppSession>
        aClass, Object... args) throws InternalException;

    void registerAppFacory(Class<? extends AppSession> sessionClass,
        IAppSessionFactory factory);

    void unRegisterAppFacory(Class<? extends AppSession> sessionClass);

    IConcurrentFactory getConcurrentFactory();

}
```

`RawSession getNewRawSession() throws InternalException;`

This method creates a `RawSession`. Raw sessions are meant as handles for code performing part of the routing decision on the stack's, such as rely agents for instance.

`Session getNewSession() throws InternalException;`

This method creates a session that acts as the endpoint for peer communication (for a given session ID). It declares the method that works with the `Request` and `Answer` objects. A session created with this method has an autogenerated ID. It should be considered as a client session.

`Session getNewSession(String sessionId) throws InternalException;`

As above. However, the created session has an ID equal to that passsed as an argument. This created session should be considered a server session.

`<T extends AppSession> T getNewAppSession(ApplicationId applicationId, Class<? extends AppSession> userSession) throws InternalException;`

This method creates a new specific application session, identified by the application ID and class of the session passed. The session ID is generated by implementation. New application sessions should be considered as client sessions. It is safe to type cast the return value to class passed as an argument. This method delegates the call to a specific application session factory.

`<T extends AppSession> T getNewAppSession(String sessionId, ApplicationId applicationId, Class<? extends AppSession> userSession) throws InternalException;`

As above. However, the session Id is equal to the argument passed. New sessions should be considered server sessions.

`<T extends AppSession> T getNewAppSession(String sessionId, ApplicationId applicationId, java.lang.Class<? extends AppSession> aClass, Object… args) throws InternalException;`

As above. However, it allows the stack to pass some additional arguments. Passed values are implementation specifc.

`void registerAppFacory(Class<? extends AppSession> sessionClass, IAppSessionFactory factory);`

Registers the `factory` for a certain `sessionClass`. This factory will receive a delegated call when ever the `getNewAppSession` method is called with an application class matching one from the register method.

`void unRegisterAppFacory(Class<? extends AppSession> sessionClass);`

Removes the application session factory registered for the `sessionClass`.

Example 1. SessionFactory use example

```
class Test implements EventListener<Request, Answer>
{

....
public void test(){
    Stack stack = new StackImpl();
    XMLConfiguration config = new XMLConfiguration(new FileInputStream(new File
(configFile));

    SessionFactory sessionFactory = stack.init(config);
    stack.start();
    //perferctly legal, both factories are the same.
    sessionFactor = stack.getSessionFactory();
    Session session = sessionFactory.getNewSession();
    session.setRequestListener(this);
    Request r = session.createRequest(308,ApplicationId.createByAuth(100L,10101L),
        "mobicents.org","aaa://uas.fancyapp.mobicents.org");

    //add avps specific for app
    session.send(r,this);
    }
}
```

Example 2. SessionFactory use example

```
class Test implements EventListener<Request, Answer>
{
    Stack stack = new StackImpl();
    XMLConfiguration config = new XMLConfiguration(new FileInputStream(new File
(configFile));

    ISessionFactory sessionFactory = (ISessionFactory)stack.init(config);
    stack.start();
    //perferctly legal, both factories are the same.
    sessionFactor = (ISessionFactory)stack.getSessionFactory();
    sessionFactory.registerAppFacory(ClientShSession.class, new
ShClientSessionFactory(this));

    //our implementation of factory does not require any parameters
    ClientShSession session = (ClientShSession) sessionFactory.getNewAppSession
(null, null
        , ClientShSession.class, null);

    ...
    session.sendUserDataRequest(udr);
}
```

Sessions

RawSessions, Sessions and ApplicationSessions provide the means for dispatching and receiving messages. Specific implementation of ApplicationSession may provide non standard methods.

The RawSession and the Session life span is controlled entirely by the application. However, the ApplicationSession life time depends on the implemented state machine.

RawSession is defined as follows:

```
public interface BaseSession extends Wrapper, Serializable {

    long getCreationTime();

    long getLastAccessedTime();

    boolean isValid();

    Future<Message> send(Message message) throws InternalException,
        IllegalDiameterStateException, RouteException, OverloadException;

    Future<Message> send(Message message, long timeOut, TimeUnit timeUnit)
        throws InternalException, IllegalDiameterStateException, RouteException,
OverloadException;

    void release();
}

public interface RawSession extends BaseSession {

    Message createMessage(int commandCode, ApplicationId applicationId, Avp... avp);

    Message createMessage(int commandCode, ApplicationId applicationId,
        long hopByHopIdentifier, long endToEndIdentifier, Avp... avp);

    Message createMessage(Message message, boolean copyAvps);

    void send(Message message, EventListener<Message, Message> listener)
        throws InternalException, IllegalDiameterStateException, RouteException,
OverloadException;

    void send(Message message, EventListener<Message, Message> listener,
        long timeOut, TimeUnit timeUnit) throws InternalException,
        IllegalDiameterStateException, RouteException, OverloadException;
}
```

`long getCreationTime();`
: Returns the time stamp of this session creation.

`long getLastAccessedTime();`
: Returns the time stamp indicating the last sent or received operation.

`boolean isValid();`
: Returns `true` when this session is still valid (ie, `release()` has not been called).

`void release();`
: Application calls this method to inform the user that the session should free any associated resource - it shall not be used anymore.

`Future<Message> send(Message message)`

Sends a message in async mode. The `Future` reference provides the means of accessing the answer once it is received

`void send(Message message, EventListener<Message, Message> listener, long timeOut, TimeUnit timeUnit)`

As above. Allows to specify the time out value for send operations.

`Message createMessage(int commandCode, ApplicationId applicationId, Avp… avp);`

Creates a Diameter message. It should be explicitly set either as a request or answer. Passed parameters are used to build messages.

`Message createMessage(int commandCode, ApplicationId applicationId, long hopByHopIdentifier, long endToEndIdentifier, Avp… avp);`

As above. However, it also allows for the Hop-by-Hop and End-to-End Identifiers in the message header to be set. This method should be used to create answers.

`Message createMessage(Message message, boolean copyAvps);`

Clones a message and returns the created object. The copyAvps parameter defines whether basic AVPs (Session, Route and Proxy information) should be copied to the new object.

`void send(Message message, EventListener<Message, Message> listener)`

Sends a message. The answer will be delivered by the specified listener

`void send(Message message, EventListener<Message, Message> listener, long timeOut, TimeUnit timeUnit)`

As above. It also allows for the answer to be passed after timeout.

`Session` defines similar methods, with exactly the same purpose:

```
public interface Session extends BaseSession {
    String getSessionId();

    void setRequestListener(NetworkReqListener listener);

    Request createRequest(int commandCode, ApplicationId appId, String destRealm);

    Request createRequest(int commandCode, ApplicationId appId, String destRealm,
String destHost);

    Request createRequest(Request prevRequest);

    void send(Message message, EventListener<Request, Answer> listener)
        throws InternalException, IllegalDiameterStateException, RouteException,
OverloadException;

    void send(Message message, EventListener<Request, Answer> listener, long timeOut,
        TimeUnit timeUnit) throws InternalException, IllegalDiameterStateException,
        RouteException, OverloadException;
}
```

Application Session Factories

In the table below, you can find session factories provided by current implementation, along with a short description:

Table 1. Application Factories

Factory class	Application type & id	Application	Reference
org.jdiameter.common.impl.app.acc.AccSessionFactoryImpl	AccountingId[0:3]	Acc	FC3588
org.jdiameter.common.impl.app.auth.AuthSessionFactoryImpl	Specific	Auth	RFC3588
org.jdiameter.common.impl.app.cca.CCASessionFactoryImpl	AuthId[0:4]	CCA	RFC4006
org.jdiameter.common.impl.app.sh.ShSessionFactoryImpl	AuthId[10415:16777217]	Sh	TS.29328, TS.29329
org.jdiameter.common.impl.app.cxdx.CxDxSessionFactoryImpl	AuthId[13019:16777216]	Cx	TS.29228, TS.29229
org.jdiameter.common.impl.app.cxdx.CxDxSessionFactoryImpl	AuthId[10415:16777216]	Dx	TS.29228, TS.29229
org.jdiameter.common.impl.app.acc.AccSessionFactoryImpl	AccountingId[10415:3]	Rf	S.32240
org.jdiameter.common.impl.app.cca.CCASessionFactoryImpl	AuthId[10415:4]	Ro	TS.32240

There is no specific factory for Ro and Rf. Those applications reuse the respective session and session factories.

Application IDs contain two numbers - [VendorId:ApplicationId].

Spaces have been introduced in the `Factory class` column values to correctly render the table. Please remove them when using copy/paste.