

Atmosphere Pro



White Paper

V 2.5.3 @ February 2017

@Copyright 2013-2017 Async-IO.org

Atmosphere Satellite	2
How to install Satellite	3
Reaching the Maximum Atmosphere Pro Licenses	4
Configuring Satellite using Hazelcast's hazelcast.xml	5
Configuring Satellite using Hazelcast using web.xml only	5
Configuring Satellite using external Hazelcast Support	6
Configuring Satellite using external Hazelcast Client	7
How replication works	8
Broadcaster Replication	9
AtmosphereResource Replication	10
AtmosphereResource Retrieval	11
Message Replication	12
Failing to deliver a message	13
Configuring Satellite Receiver Thread Pool for optimal performance	14
Atmosphere Tower Control	15
Installing Tower Control	15
Starting Tower Control	16
<i>io.async.control.AsyncSupport</i>	18
<i>io.async.control.AtmosphereFramework</i>	19
<i>io.async.control.broadcaster</i>	20
<i>io.async.control.cache</i>	20
<i>io.async.control.config</i>	21
<i>io.async.control.factory</i>	21
<i>io.async.control.interceptors</i>	22
<i>io.async.control.resource</i>	22
<i>io.async.control.statistics</i>	23
<i>io.async.control.websocket</i>	23
Atmosphere Postman	24
How to install Postman	24
Installing the client side	25
<i>Callbacks</i>	25
<i>How it works</i>	25

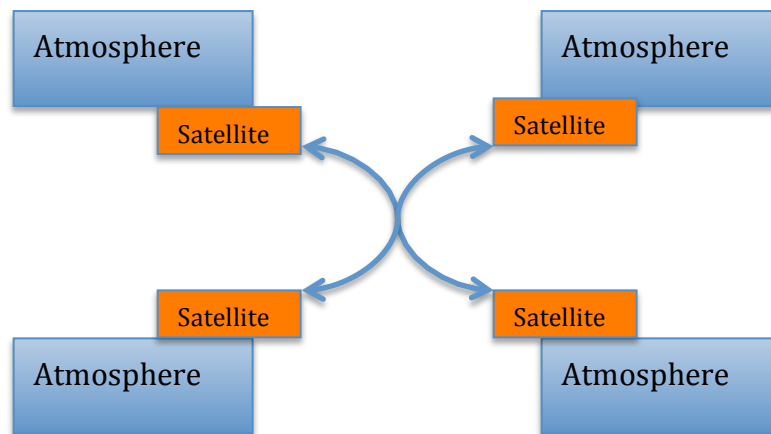
Atmosphere Satellite

Applications using Atmosphere Satellite gain the following superpowers:

- Elastic Scalability (just add more servers and they cluster auto magically, i.e. automatically)
- Super Speeds (in memory transaction speeds)
- High Availability
- Fault Tolerance
- Cloud Readiness
- 100% State Replication of AtmosphereResource, Broadcaster and BroadcasterCache
- Powered by Hazelcast 3.x (www.hazelcast.org): An In-Memory Data Grid

Atmosphere Satellite is easily able to handle this type of use case with in-memory performance, linear scalability as you add new nodes and reliability.

Satellite is small and gets transparently enabled by Atmosphere. Because of its design approach as well as Satellite's ability to automatically discover and cluster with peers, Satellite provides drop-in session clustering ability for any Atmosphere enabled server. It requires no additional investment in hardware and elastically scales as you add Atmosphere's application. This is a great way to ensure that Atmosphere's session state is maintained when you are clustering Atmosphere servers.



How to install Satellite

To install Satellite, all you need to do is to add the following dependency in your pom.xml:

```
<dependency>
  <groupId>io.async</groupId>
  <artifactId>atmosphere-satellite</artifactId>
  <version>2.5.3</version>
</dependency>
```

Atmosphere will auto-detect the jar and will install Satellite automatically. Once installed, you should see in your log:

```
Sep 12, 2014 11:02:17 AM com.hazelcast.cluster.MulticastJoiner
INFO: [10.0.1.4]:5701 [dev]
Members [1] {
  Member [10.0.1.4]:5701 this
}
Sep 12, 2014 11:02:17 AM com.hazelcast.core.LifecycleService
INFO: [10.0.1.4]:5701 [dev] Address[10.0.1.4]:5701 is STARTED
11:02:17.098 INFO [main] i.a.s.Satellite [Satellite.java:81]
Atmosphere Satellite e6ba40a1-29ab-493c-9295-30bcc7a2aaea
11:02:17.106 INFO [main] o.a.c.AtmosphereFramework [AtmosphereFramework.java:1906]
Auto detecting WebSocketHandler in /WEB-INF/classes/
```

When another Atmosphere Satellite instance is getting installed, the log for the first installation will show

```
Sep 12, 2014 11:14:31 AM com.hazelcast.cluster.ClusterService
INFO: [10.0.1.4]:5701 [dev]
Members [2] {
  Member [10.0.1.4]:5701 this
  Member [10.0.1.4]:5702
}
}
```

Addition and removal of Atmosphere Satellite are dynamic and will always be reflected in the log of each Atmosphere Satellite installation. You can also browse Satellite's MBeans using your favorite JMX client under the 'io.async.satellite' package.

Reaching the Maximum Atmosphere Pro Licenses

If you install more Atmosphere Pro instances than the number bought, any new instance will throw

```
io.async.satellite.Satellite$MaxLicensesException: Maximum Licenced Satellite 27  
at io.async.satellite.Satellite.dispatchMessage(Satellite.java:164)  
at io.async.satellite.Satellite.access$100(Satellite.java:46)  
at io.async.satellite.Satellite$2.on(Satellite.java:79)  
at  
io.async.satellite.HazelcastSatelliteTransport$1.onMessage(HazelcastSatelliteTransport.java:  
43)  
at com.hazelcast.topic.impl.TopicService.dispatchEvent(TopicService.java:135)  
at  
com.hazelcast.spi.impl.EventServiceImpl$EventPacketProcessor.process(EventServiceImpl.ja  
va:545)  
at  
com.hazelcast.spi.impl.EventServiceImpl$RemoteEventPacketProcessor.run(EventServiceIm  
pl.java:625)  
at com.hazelcast.util.executor.StripedExecutor$Worker.process(StripedExecutor.java:189)  
at com.hazelcast.util.executor.StripedExecutor$Worker.run(StripedExecutor.java:173)
```

Please contact licenses@async-io.org for more licenses.

Configuring Satellite using Hazelcast's hazelcast.xml

You can configure Hazelcast by following the normal way, as recommended by the Hazelcast team

<http://hazelcast.org/docs/latest/manual/html/config.html>

By default, a Hazelcast instance named "AtmosphereSatellite" will be created. If your application already use a HazelcastInstance or want to create a new instance with a different name, just define in web/atmosphere.xml:

```
<init-param>
  <param-name>io.async.satellite.HazelcastSatelliteTransport.instanceName</param-name>
  <param-value><<name>></param-value>
</init-param>
```

If the HazelcastInstance name exists, it will be picked and if not, created. It is strongly recommended to configure Hazelcast instance via **hazelcast.xml**

Configuring Satellite using Hazelcast using web.xml only

If your application only use Hazelcast TCP/IP configuration, you can also configure it directly using the following properties

```
<init-param>
  <param-name>io.async.satellite.HazelcastSatelliteTransport.tcpIp.enabled</param-name>
  <param-value>true</param-value>
</init-param>

<init-param>
  <param-name>io.async.satellite.HazelcastSatelliteTransport.tcpIp.members</param-name>
  <param-value>5701</param-value>
</init-param>

<init-param>
  <param-name>io.async.satellite.HazelcastSatelliteTransport.group.name</param-name>
  <param-value>opd-poc</param-value>
</init-param>

<init-param>
  <param-name>io.async.satellite.HazelcastSatelliteTransport.group.password</param-name>
  <param-value>opd-poc-pass</param-value>
```

Configuring Satellite using external Hazelcast Support

You can configure Satellite's internal Hazelcast instance by implementing an HazelcastConfigurator:

```
package io.async.satellite;

import com.hazelcast.core.HazelcastInstance;
import org.atmosphere.cpr.AtmosphereConfig;

public interface HazelcastConfigurator {

    HazelcastInstance getOrCreateHazelcastInstance(AtmosphereConfig config);

}
```

and by defining

```
<init-param>
  <param-name>io.async.satellite.HazelcastConfigurator.className</param-name>
  <param-value>xxxxx</param-value>
</init-param>
```

Configuring Satellite using external Hazelcast Client

If you already have a running Hazelcast instance, you can also configure Satellite to use it instance by adding to your configuration file:

```
<init-param>
  <param-name>io.async.satellite.HazelcastSatelliteTransport.useClient</param-name>
  <param-value>true</param-value>
</init-param>
<init-param>
  <param-name>io.async.satellite.HazelcastSatelliteTransport.address</param-name>
  <param-value>127.0.0.1:5701</param-value>
</init-param>
```


How replication works

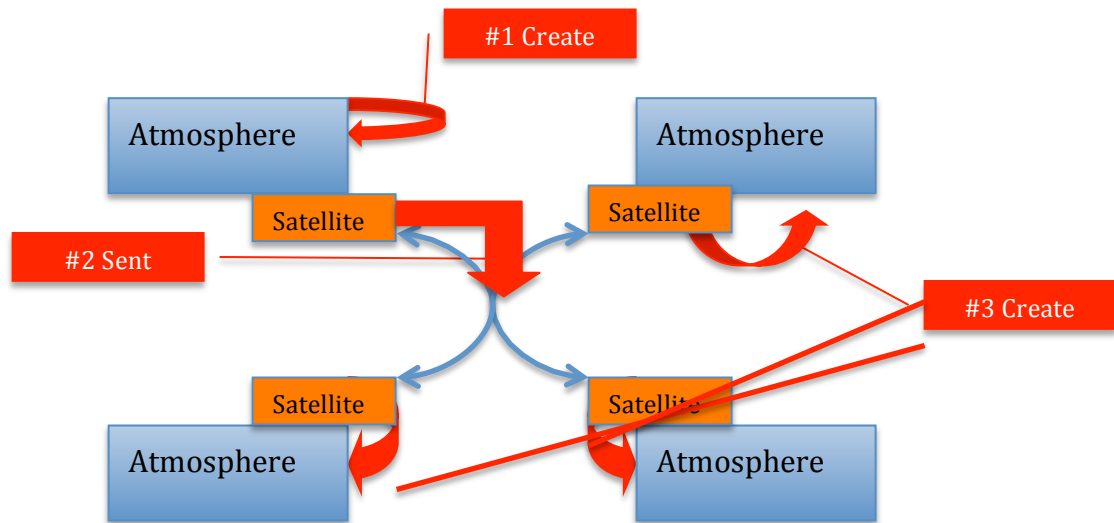
A Satellite always publishes its state at the moment it changes. Changing states includes:

- **Broadcaster:** when a Broadcaster is created by one Satellite, a message will be sent to all others Satellites, asking them to create the Broadcaster. Hence all available Satellites will have the same set of created Broadcasters. The same will happen when a Broadcaster is destroyed from one of the Satellite, e.g. the removal will also be execute by all Atmosphere Satellite.
- **AtmosphereResource:** when an AtmosphereResource is created (when a users/browsers connects), a message will be sent to all others Satellites, asking them to register the AtmosphereResource's UUID with its associated broadcaster. When an AtmosphereResource gets registered with a Broadcaster, the AtmosphereResource becomes candidate for message caching. The same will happens when an AtmosphereResource gets removed.
- **BroadcasterCache:** BroadcasterCache are tightly coupled with Broadcaster. Hence, BroadcasterCache are getting created everytime a Broadcaster is. Every time a new AtmosphereResource is added to a Broadcaster, a message will be sent to all others Satellites, and the AtmosphereResource's UUID will be added to the list of active BroadcasterCache's.

Broadcaster Replication

As noted, Broadcaster are getting replicated by following:

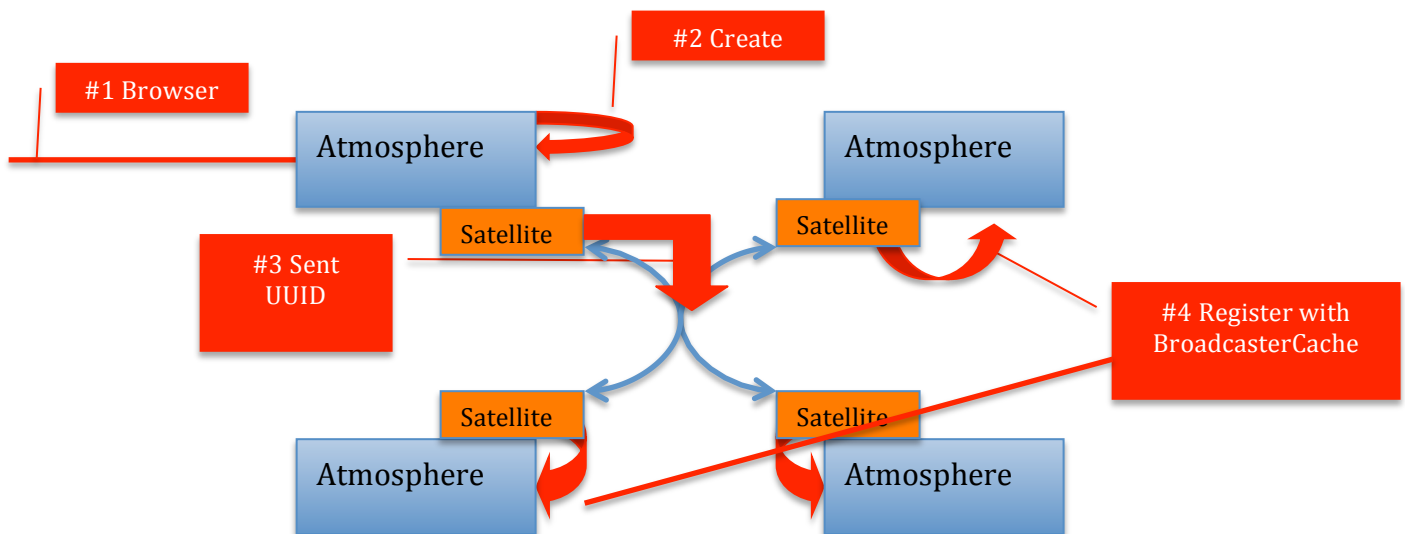
1. In a Satellite, a Broadcaster is getting created (#1)
2. The Satellite send a message to other Satellites (#2)
3. The Broadcaster is created in the other Satellite (#3)



AtmosphereResource Replication

When a user/browser connect, an AtmosphereResource is always created. With Satellite, the UUID of that AtmosphereResource will be shared amongst the Satellites:

1. Browser connects
2. AtmosphereResource gets created
3. The Atmosphere's UUID is sent to all Satellites
4. The UUID is registered with BroadcasterCache. Registering UUID with BroadcasterCache means message will be cached for that resource unless once of the Satellite successfully deliver the message.



AtmosphereResource Retrieval

It is possible to retrieve an AtmosphereResource located on another node by using the *AtmosphereResourceFactory.locate(uuid, Async)* API. Under the hood class *AtmosphereResourceFactory* will communicate with the remote Satellites and create a local “stub” for the remote AtmosphereResource if located.

```
factory.locate(message.getMessage(), new AtmosphereResourceFactory.Async() {  
    @Override  
    public void available(AtmosphereResource r) {  
        // Do something with the resource  
        r.write( "Hello World" );  
    }  
});
```

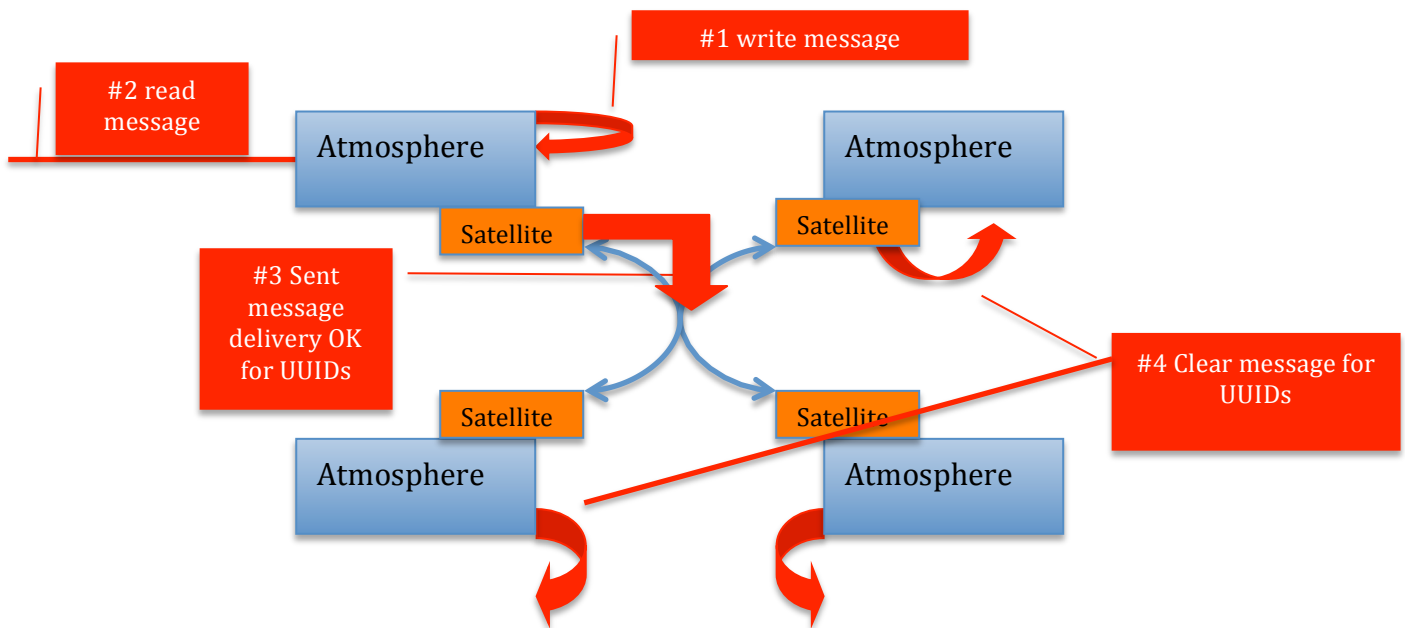
The operation of locating an AtmosphereResource is asynchronous, and it is left to the application developer to block in case of the retrieval must be done synchronously:

```
final AtomicReference<AtmosphereResource> resource  
    = new AtomicReference<AtmosphereResource>();  
final CountDownLatch latch = new CountDownLatch(1);  
  
factory.locate(message.getMessage(), new AtmosphereResourceFactory.Async() {  
    @Override  
    public void available(AtmosphereResource r) {  
        resource.set(r);  
        latch.countDown();  
    }  
});  
latch.await();  
  
AtmosphereResource stub = resource.get();  
stub.write("Hello World");
```

Message Replication

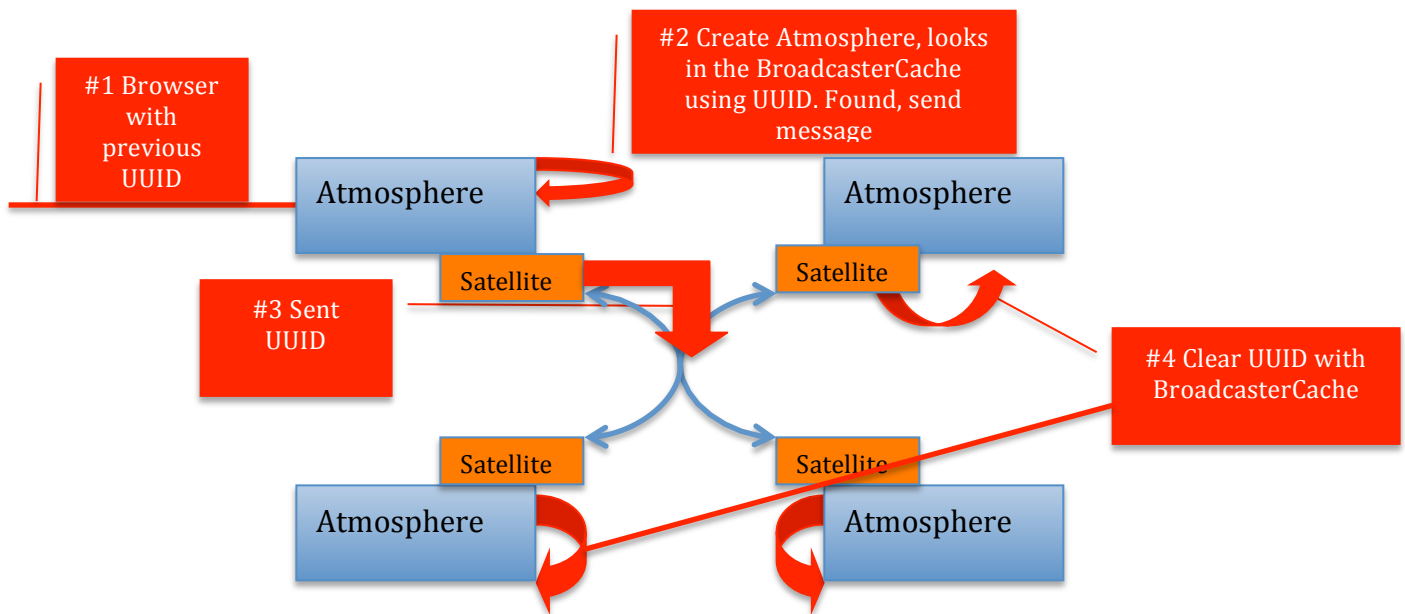
When a message is broadcasted in one Satellite, the message will be sent to all Satellites, which in turn will broadcast to their set of AtmosphereResources and cached for AtmosphereResource not located on that Satellite.

As soon as the message is successfully delivered in one Satellite, the information will be sent to all other Satellites so they can remove the message from their BroadcasterCache.



Failing to deliver a message

If, for any reason the message is not delivered to the AtmosphereResource, the message will stay in all Satellite's BroadcasterCache so when the browser reconnects using its previous AtmosphereResource's UUID, the message will be pulled from the cache and send. Then all Satellites will be advised the message has been delivered and



Configuring Satellite Receiver Thread Pool for optimal performance

If your application sends and receives thousand of messages, it is important to properly configure the Satellite's receiver thread pool size:

```
<init-param>  
  <param-name>io.async.satellite.HazelcastSatelliteTransport.threadPoolSize</param-name>  
  <param-value>200</param-value>  
</init-param>
```

By default, the number of available processors will be used.

Atmosphere Tower Control

Atmosphere Tower Control is designed with ease of use and flexibility in mind and delivers unprecedented power to Atmosphere's Developers. Atmosphere Tower Control is a complete ecosystem for developers, offering a complete end to end solution for monitoring and debugging an Atmosphere application.

Tower Control offers the ability to completely configure Atmosphere, collect statistics, reload Atmosphere applications and hot swap an Atmosphere applications remotely.

Installing Tower Control

To install Satellite, all you need to do is to add the following dependency in your pom.xml:

```
<dependency>
  <groupId>io.async</groupId>
  <artifactId>atmosphere-tower-control</artifactId>
  <version>2.4.6</version>
</dependency>
```

Atmosphere will auto-detect the jar and will install Tower Control automatically. Once installed, you should see in your log:

```
11:56:17.411 INFO [main] o.a.c.AnnotationHandler [AnnotationHandler.java:63] Found
Annotation in class io.async.control.TowerControlInterceptor being scanned: interface
org.atmosphere.config.service.AtmosphereInterceptorService
```

```
11:56:17.537 INFO [main] i.a.c.TowerControlInterceptor [TowerControlInterceptor.java:58]
```

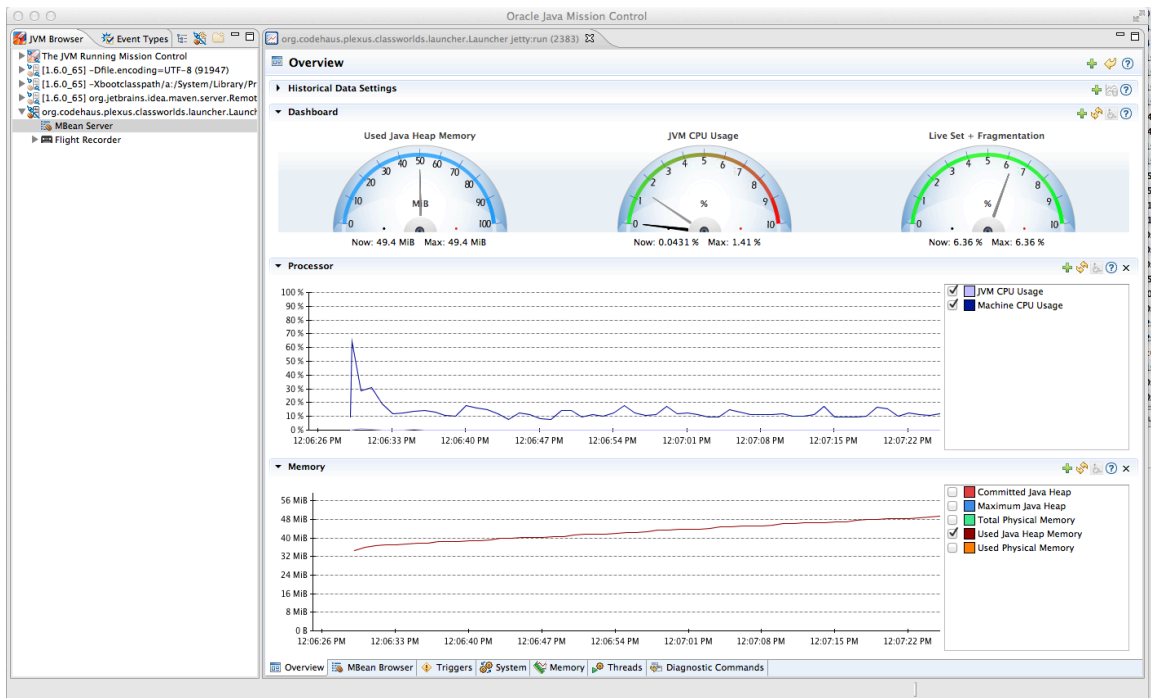
Atmosphere Tower Control

```
11:56:17.538 INFO [main] o.a.c.AnnotationHandler [AnnotationHandler.java:63] Found
Annotation in class io.async.control.TowerControlListener being scanned: interface
org.atmosphere.config.service.BroadcasterListenerService
```


Starting Tower Control

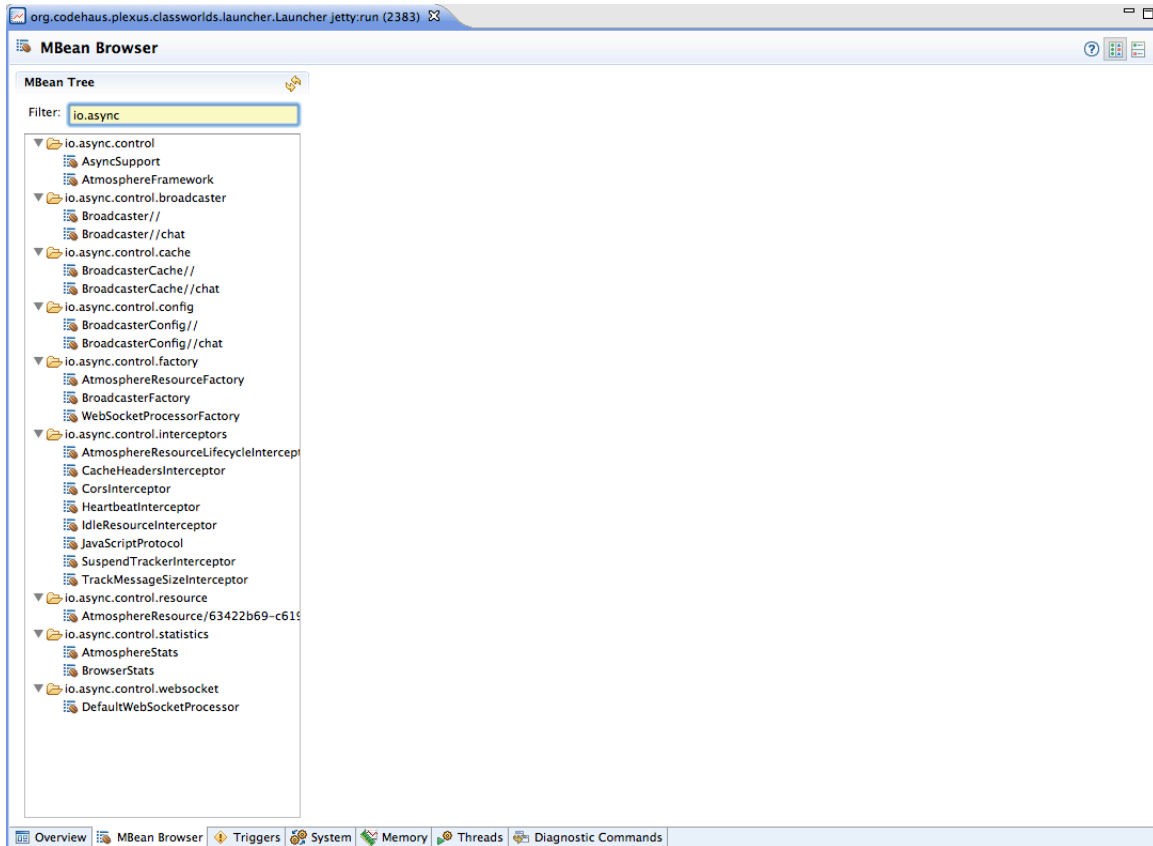
To start Tower Control, all you need to have is a tool supporting JMX. For example, both Java Mission Control(JMC) and jVisualVM supports JMX. Let's assume we will use JMC. To start JMC, just do:

```
% jmc
```



Next, select under the JVM Browser Tab, select the Java process, which started your Atmosphere Application. On the right side, select the MBean Tab. You should see

To make things simpler, type `io.async` in the Filter field, so we just see Tower Control Beans



All the Atmosphere's MBeans are grouped by type:

- **io.async.control:** Contains information about installed `AsyncSupport` and `AtmosphereFramework` classes.
- **io.async.control.broadcaster:** The current set of created `Broadcaster` with their associated state.
- **io.async.control.cache:** The current set of created `BroadcasterCache` with their associated state.
- **io.async.control.factory:** The `BroadcasterFactory`, `AtmosphereResourceFactory` and `WebSocketProcessorFactory` with their associated state.
- **io.async.control.interceptors:** The current set of installed `AtmosphereInterceptor` with their associated state.
- **io.async.control.resource:** The current set of connected clients, represented by their `AtmosphereResource`.
- **io.async.statistics:** Live statistic like number of connections, messages, transport used as well as Browser's used.
- **io.async.control.websocket:** Contains information about installed `WebSocketProcessor`

Let's explore them one by one and see what kind of information is available from those beans.

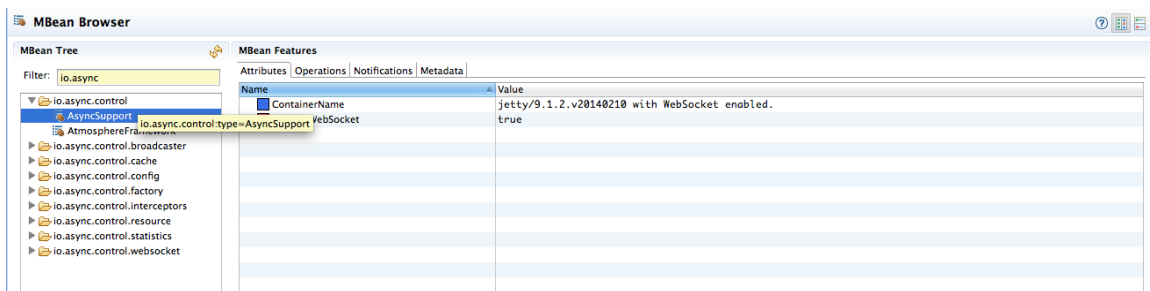
`io.async.control.AsyncSupport`

Attributes

Contains information about the server used and if websocket is supported or not.

Operations:

No operation available



The screenshot shows the MBean Browser interface. On the left, the MBean Tree is filtered by 'io.async'. The 'io.async.control.AsyncSupport' MBean is selected. On the right, the MBean Features tab is active, showing a table of attributes.

Name	Value
ContainerName	jetty/9.1.2.v20140210 with WebSocket enabled.
WebSocket	true

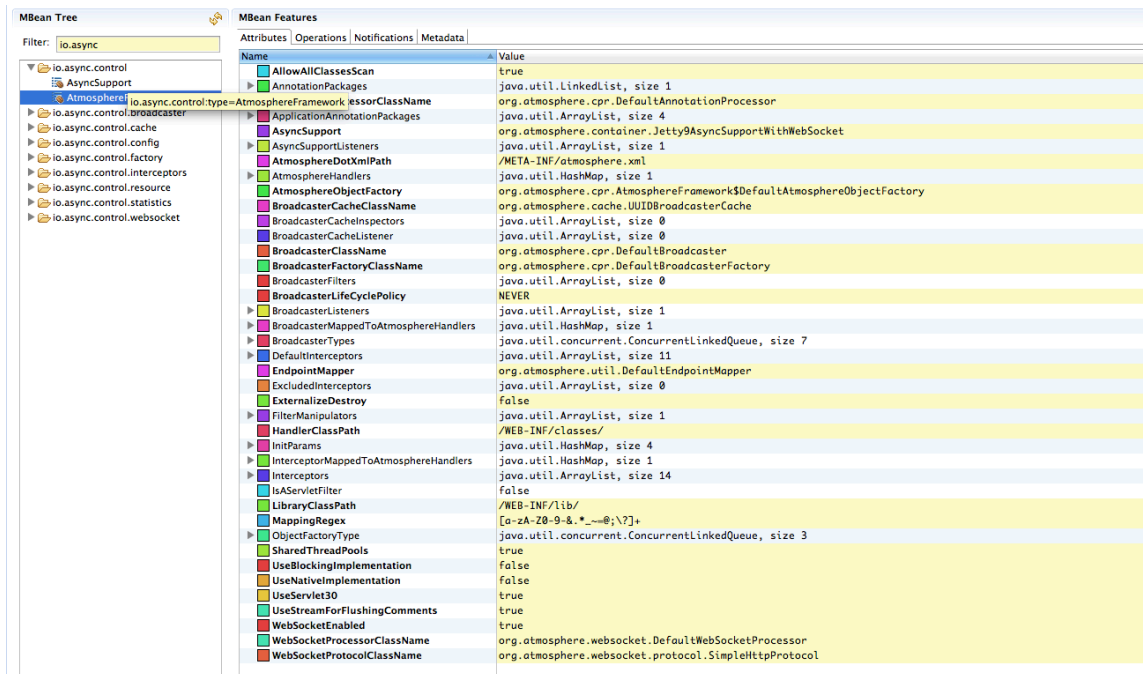
io.async.control.AtmosphereFramework

Attributes

Contains information about how Atmosphere has been started and configured. Everything configured by default or via web/application.xml is getting reflected.

Operations:

You can reconfigure the AtmosphereFramework via the MBean's operations. For example, you can add BroadcasterListener, init-params etc. and then click on reload to reconfigure the AtmosphereFramework.



Name	Value
AllowAllClassesScan	true
AnnotationPackages	java.util.LinkedList, size 1
ApplicationAnnotationPackages	org.atmosphere.cpr.DefaultAnnotationProcessor
AsyncSupport	java.util.ArrayList, size 4
AsyncSupportListeners	org.atmosphere.container.Jetty9AsyncSupportWithWebSocket
AtmosphereDotXmlPath	java.util.ArrayList, size 1
AtmosphereHandlers	/META-INF/atmosphere.xml
AtmosphereObjectFactory	java.util.HashMap, size 1
BroadcasterCacheClassName	org.atmosphere.cpr.AtmosphereFramework\$DefaultAtmosphereObjectFactory
BroadcasterCacheInspectors	org.atmosphere.cache.UUIDBroadcasterCache
BroadcasterCacheListener	java.util.ArrayList, size 0
BroadcasterClassName	org.atmosphere.cpr.DefaultBroadcaster
BroadcasterFactoryClassName	org.atmosphere.cpr.DefaultBroadcasterFactory
BroadcasterFilters	java.util.ArrayList, size 0
BroadcasterLifeCyclePolicy	NEVER
BroadcasterListeners	java.util.ArrayList, size 1
BroadcasterMappedToAtmosphereHandlers	java.util.HashMap, size 1
BroadcasterTypes	java.util.concurrent.ConcurrentLinkedQueue, size 7
DefaultInterceptors	java.util.ArrayList, size 11
EndpointMapper	org.atmosphere.util.DefaultEndpointMapper
ExcludedInterceptors	java.util.ArrayList, size 0
ExternalizeDestroy	false
FilterManipulators	java.util.ArrayList, size 1
HandlerClassPath	/WEB-INF/classes/
InitParams	java.util.HashMap, size 4
InterceptorMappedToAtmosphereHandlers	java.util.HashMap, size 1
Interceptors	java.util.ArrayList, size 14
IsAServletFilter	false
LibraryClassPath	/WEB-INF/lib/
MappingRegex	[a-zA-Z0-9-@.*_-~@:~?~+]
ObjectFactoryType	java.util.concurrent.ConcurrentLinkedQueue, size 3
SharedThreadPools	true
UseBlockingImplementation	false
UseNativeImplementation	false
UseServlet30	true
UseStreamForFlushingComments	true
WebSocketEnabled	true
WebSocketProcessorClassName	org.atmosphere.websocket.DefaultWebSocketProcessor
WebSocketProtocolClassName	org.atmosphere.websocket.protocol.SimpleHttpProtocol

io.async.control.broadcaster

Attributes

Contains information about the current state of a Broadcaster. For example, the AtmosphereResource, the number of them, the installed BroadcasterListener, etc.

Operations:

Several operations are available, like adding, on the fly, AtmosphereResource, broadcasting messages, resuming AtmosphereResources etc.

The screenshot shows the MBean Tree on the left with the filter 'io.async'. The 'io.async.control.broadcaster' MBean is selected. The MBean Features table on the right displays the following attributes:

Name	Value
AtmosphereResourceCount	1
AtmosphereResources	java.util.ArrayList, size 1
BroadcasterListeners	java.util.ArrayList, size 2
destroyed	false
/chat	/chat
LifeCycleListeners	java.util.ArrayList, size 0
LifeCyclePolicy	NEVER
Messages	java.util.ArrayList, size 0
OutOfOrderBroadcastSupported	false
Policy	FIFO
Scope	APPLICATION
WriteQueues	java.util.concurrent.ConcurrentHashMap, size 0

io.async.control.cache

Attributes

Contains information about the current state of the BroadcasterCache like active AtmosphereResource, installed listeners etc.

Operations:

Message can be added,excluded or deleted from the cache

The screenshot shows the MBean Tree on the left with the filter 'io.async'. The 'io.async.control.cache' MBean is selected. The MBean Features table on the right displays the following attributes:

Name	Value
ActiveClients	java.util.concurrent.ConcurrentHashMap, size 1
63422b69-c619-415b-ae09-58be3d2a3f74	1410280193577
/elInspectors	java.util.ArrayList, size 0
/elListeners	java.util.ArrayList, size 0
Messages	java.util.HashMap, size 0

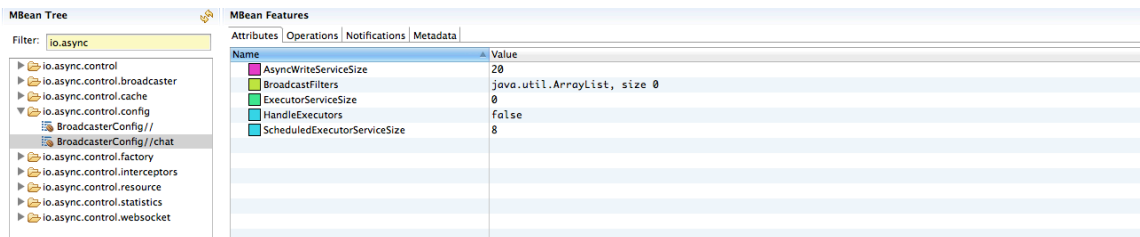
io.async.control.config

Attributes

Contains information about the BroadcasterConfig likes Thread, Thread Pool, etc.

Operations:

You can destroy or remove BroadcastFilter



The screenshot shows the JMX console with the MBean Tree on the left and the MBean Features table on the right. The filter is set to 'io.async'. The MBean 'io.async.control.config' is selected, and its attributes are displayed in the table.

Name	Value
AsyncWriteServiceSize	20
BroadcastFilters	java.util.ArrayList, size 0
ExecutorServiceSize	0
HandleExecutors	false
ScheduledExecutorServiceSize	8

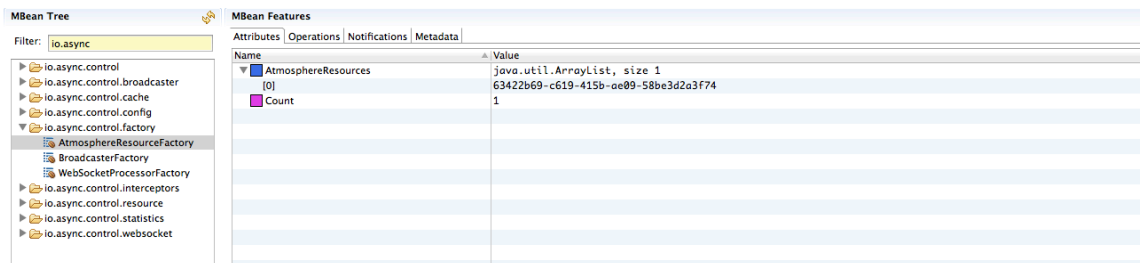
io.async.control.factory

Attributes

Contains information about the number of AtmosphereResource, Broadcaster and WebSocketProcessor

Operations:

You add find AtmosphereResource based on their UUID



The screenshot shows the JMX console with the MBean Tree on the left and the MBean Features table on the right. The filter is set to 'io.async'. The MBean 'io.async.control.factory' is selected, and its attributes are displayed in the table.

Name	Value
AtmosphereResources	java.util.ArrayList, size 1
[0]	63422b69-c619-415b-ae09-58be3d2a3f74
Count	1

io.async.control.interceptors

Attributes

Contains information about the installed AtmosphereInterceptors

Operations:

For example, you can configure on the fly the SuspendTrackerInterceptor.

The screenshot shows the JBoss JMX console. On the left, the 'MBean Tree' panel displays a hierarchy of MBeans. The 'io.async.control.interceptors' MBean is expanded, showing several sub-MBeans including 'SuspendTrackerInterceptor'. On the right, the 'MBean Features' panel is active, showing the 'Operations' tab for the selected MBean. The operations listed are 'addTrackedUUID : void', 'clear : void', and 'removeTrackedUUID : void'. The 'Name', 'Value', and 'Description' columns are empty.

io.async.control.resource

Attributes

The list of current connected users, or AtmosphereResource.

Operations:

You can close and or resume an existing AtmosphereResource

The screenshot shows the JBoss JMX console. On the left, the 'MBean Tree' panel displays a hierarchy of MBeans. The 'io.async.control.resource' MBean is expanded, showing several sub-MBeans including 'AtmosphereResource'. On the right, the 'MBean Features' panel is active, showing the 'Attributes' tab for the selected MBean. The attributes listed are 'AsyncWriter', 'AtmosphereHandler', 'Attributes', 'Broadcaster', 'Cancelled', 'Headers', 'Listeners', 'QueryString', 'RequestDestroyed', 'ResponseDestroyed', 'Resumed', 'Serializer', 'Suspended', 'Transport', 'UserAgent', and 'Uuid'. The 'Value' column shows the values for these attributes, and the 'Update' column shows the update time for each attribute.

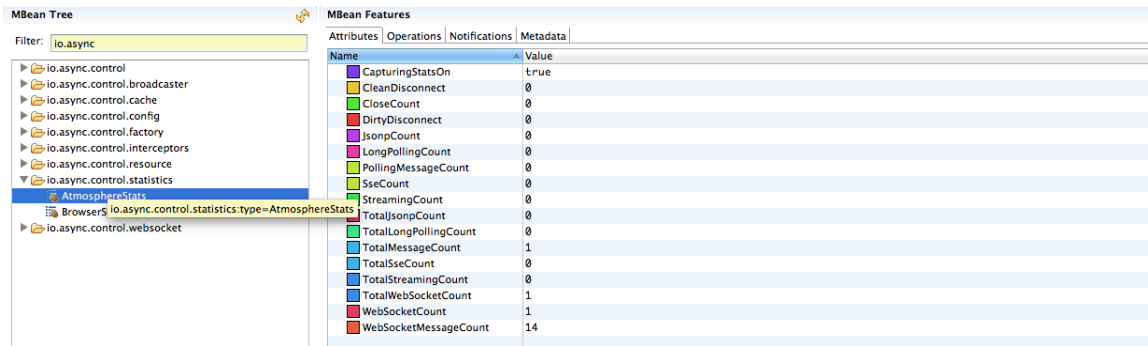
io.async.control.statistics

Attributes

Collect statistics about the current number of connections, disconnected count, total received messages etc.

Operations:

Enable/Disable statistics collection.



The screenshot shows the JBoss JMX console with the MBean Tree on the left and the MBean Features table on the right. The filter is set to 'io.async'. In the MBean Tree, 'io.async.control.statistics' is selected, and its sub-attribute 'AtmosphereStats' is expanded. The MBean Features table displays various statistics attributes and their current values.

Name	Value
CapturingStatsOn	true
CleanDisconnect	0
CloseCount	0
DirtyDisconnect	0
JsoupCount	0
LongPollingCount	0
PollingMessageCount	0
SseCount	0
StreamingCount	0
TotalJsoupCount	0
TotalLongPollingCount	0
TotalMessageCount	1
TotalSseCount	0
TotalStreamingCount	0
TotalWebSocketCount	1
WebSocketCount	1
WebSocketMessageCount	14

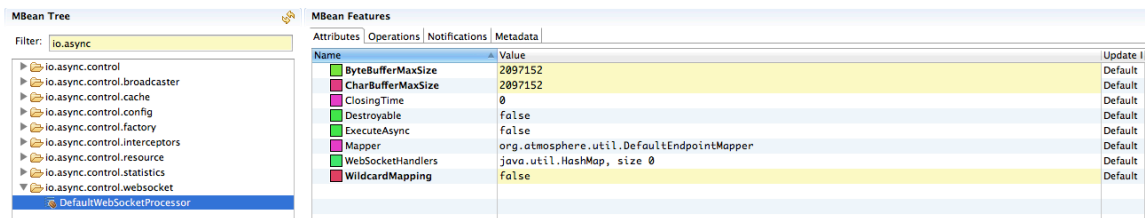
io.async.control.websocket

Attributes

Information about how websockets are installed and configured

Operations:

None



The screenshot shows the JBoss JMX console with the MBean Tree on the left and the MBean Features table on the right. The filter is set to 'io.async'. In the MBean Tree, 'io.async.control.websocket' is selected, and its sub-attribute 'DefaultWebSocketProcessor' is expanded. The MBean Features table displays various configuration attributes and their current values.

Name	Value	Update I
ByteBufferMaxSize	2097152	Default
CharBufferMaxSize	2097152	Default
ClosingTime	0	Default
Destroyable	false	Default
ExecuteAsync	false	Default
Mapper	org.atmosphere.util.DefaultEndpointMapper	Default
WebSocketHandlers	java.util.HashMap, size 0	Default
WildcardMapping	false	Default

Atmosphere Postman

With Guaranteed Delivery, the Atmosphere Postman system uses a built-in data store to persist messages. Atmosphere Postman guarantees the client that when sending a message; the message will always be delivered to the server. In case of a failure, the message will be re-sent until it reaches the server. Postman guarantee client's messages delivery.

When combined with Atmosphere Satellite, when a websocket or fallback transport delivers a message, the send operation does not complete successfully until the message is safely stored in the sender's data store. Subsequently, the message is not deleted from one data store until it is successfully forwarded to and stored in the next data store. As a result, once a websocket or fallback transport successfully sends a message, it is also stored in memory on at least one Atmosphere Satellite until the message has been successfully delivered and acknowledged by the browser. **Installing Satellite and Postman guarantee 100% messages delivery, both from the browser and the server.**

How to install Postman

To install Satellite, all you need to do is to add the following dependency in your pom.xml:

```
<dependency>
  <groupId>io.async</groupId>
  <artifactId>atmosphere-postman</artifactId>
  <version>1.0.0</version>
</dependency>
```

Atmosphere will auto-detect the jar and will install Tower Control automatically. Once installed, you should see in your log:

```
17:23:33.409 INFO [main] i.a.p.ClientAckInterceptor [ClientAckInterceptor.java:46]
```

```
Atmosphere Postman : io.async.postman.ClientAckInterceptor
```

```
17:23:33.409 INFO [main] o.a.c.AtmosphereFramework [AtmosphereFramework.java:2362]
Installed AtmosphereInterceptor io.async.postman.ClientAckInterceptor with priority
AFTER_DEFAULT
```

```
17:23:33.409 INFO [main] i.a.p.ReloadAckInterceptor [ReloadAckInterceptor.java:64]
```

```
Atmosphere Postman : io.async.postman.ReloadAckInterceptor
```

Installing the client side

Client side, you need to add to you application's main page

```
<script type="text/javascript" language="javascript" src="atmosphere.js"></script>
```

```
<script type="text/javascript" language="javascript" src="atmosphere.postman.js"></script>
```

Callbacks

You can trace and react using two client's side function

```
atmosphere.onAckSuccess = function(res) {  
    console.log("onAckSuccess");  
    console.log(res);  
};  
  
atmosphere.onAckFailed = function(req) {  
    console.log("onAckFailed");  
    console.log(req);  
};
```

How it works

If `atmosphere.postman.js` is loaded in client and `ClientAckInterceptor` is included in interceptor stack in server, when you sends a message using the `push` method, the followings will happen:

1. A JSON consisting of 'id' and 'message' is created and sent instead of the message.
2. At the same time, the timer handling ACK is set in client.
 - 3.1 If server receives it,
 - 4.1. `ClientAckInterceptor` parses that JSON, restore the message and send the ACK using the id.
 - 4.2. If the ACK is arrived in client, `atmosphere.onAckSuccess` will be executed with `AtmosphereResponse`.
 - 3.1 If server can't receive it,
 - 5.1 After `AtmosphereRequest.ackInterval` in ms or 5 seconds if it's not set, `atmosphere.onAckFailed` will be executed with `AtmosphereRequest`.
 - 5.2 At the same time, the original message is sent again then the situation goes to step 1.