R&D Workshop

Tomcat in the Cloud

Final Report

Ismaïl Senhaji
ismail.senhaji@unifr.ch

Guillaume Pythoud
guillaume.pythoud@unifr.ch

Supervised by
Jean-Frederic Clere
jclere@redhat.com

June 20, 2017

MASTER IN COMPUTER SCIENCE

redhat.

Contents

1 Introduction

1.1 Context

This project was realized for the *R&D Workshop* at the University of Neuchâtel during the spring semester 2017. We worked with Jean-Frederic Clere from *Red Hat*[1]. Our goal was to extend *Tomcat*[2] in order to make session replication and clustering work in a cloud environment such as OpenShift[3].

Until now, session replication in Tomcat was not cloud ready. The principal implementation uses multicast which only works in a local network, whereas in a cloud deployment, instances can be distributed over the internet. This report describes the solution we've developed over the course of this semester.

1.2 Goal

The major goal of the project to make Tomcat's session work in the cloud. This goal is divided in 3 steps:

1. Study existing methods
2. Implement a new solution based on one of those methods
3. Test the implementation

Another important goal is to provide detailed documentation for future users. We will write a "Quick Start Guide" detailing the steps to follow to have a working Tomcat cluster running in an OpenShift cloud.

2 Organization

All our work is available on a GitHub repository[1]. The branch *master* contains our solution: the source code of the application and a *README* file (reproduced in appendix A) containing all information necessary to configure and deploy our application. It also contains the documentation we produced during the project in the *docs* folder. The logbook of the project is accessible in the *Wiki* section.

The repository has another branch called *experiments*[2] that contains all the test applications and experiments we produced to get familiar with the new technologies. For each application there is a small *README* file that gives basics instructions for their deployment.

[1] `https://github.com/iSma/tomcat-in-the-cloud/`

[2] `https://github.com/iSma/tomcat-in-the-cloud/tree/experiments`

3 State of the art

During the initial research phase of the project, we looked for existing solutions to this problem, as well as partial solutions we could build on. We've identified 3 such paths to explore:

1. `DeltaManager`: With this built-in session manager, session data is replicated among all nodes. When using this mechanism, other nodes are discovered through multicast ping messages, or a static list of peers can be provided.

2. `PeristanceManager`: Session data is stored in any JDBC-capable database, which eliminates the need for nodes to discover each other.

3. External session management solutions: when using the *Spring Framework*[4] for instance, Tomcat's session management is bypassed and sessions are handled directly by the framework. Spring is advertised to have built-in support for data stores such as *Redis* or *Hazelcast*[3].

Solution 1 is the most attractive for the end-user, as it doesn't involve installing and running a database server, nor additional software dependencies. This is why we decided to concentrate our efforts on building a solution on top of `DeltaManager`. The main difficulty was implementing an alternative discovery process.

4 Our solution

The final solution we provide uses the built-in `DeltaManager`, but replaces the default peer discovery mechanism. Tomcat can be configured to use any class implementing the interface `MembershipService` for peer discovery. Only one such implementation is provided, `McastService`, which works by sending heartbeat messages over multicast.

We created a new implementation, `DynamicMembershipService`, which handles bookkeeping of peers and informing listeners of added and removed members. The discovery per se is delegated to a `MemberProvider`. The `DynamicMembershipService` starts a `RefreshThread`, which periodically asks the `MemberProvider` for new peers. The class hierarchy is shown in figure 1, and a detailed sequence diagram is shown in figure 2.

We provide one implementation of `MemberProvider`: `KubernetesMemberProvider`. Its operation is largely inspired by the *KubePing* project[5], a Kubernetes discovery protocol for *JGroups*. The code available in the `stream` package was reproduced from this project.

[3] `http://docs.spring.io/spring-session/docs/current/reference/html5/`

`KubernetesMemberProvider` is based on the OpenShift REST API[6], which is available to all pods[4]. We use it to get a list of all running pods. All pods automatically get injected with environment variables and files allowing them to authenticate themselves and construct the URL to access this API (see file *KubernetesMemberProvider.java* for more detail).

Accessing the path `/api/v1/namespaces/tomcat-in-the-cloud/pods` returns a JSON representation of all the pods in the deployment. An excerpt of a typical response can be found in Listing 1. This response is then parsed, and only the running pods are kept, from which we extract following data:

- `status.podIP`: The IP address of the peer
- `metadata.name`: Its host name, is md5-hashed and use as unique ID
- `metadata.creationTimestamp`: Creation time of the node

The creation time is used to compute how long the peer has been alive (relatively to the current node). This value is used internally by Tomcat: upon startup, the initial exchange of session data is made with the oldest peer.

5 Testing

During development, we needed a way to verify if session replication was working correctly. For this, we implemented a very simple test application that responds to HTTP requests with a JSON object containing following information:

- A counter that is stored in a session and is upgraded at each request
- The current session ID
- A boolean, set to `true` if the session is freshly created
- The IP address and host name of the Tomcat instance that served the request

An example output is given in Listing 2

Listing 2: Example output

```
{
    "counter": 4,
    "id": "305F63E07385E9B2E434941DF607B60E",
    "new": false,
    "server": "172.17.0.2",
    "hostname": "tomcat-in-the-cloud-1-5xbwm"
}
```

[4] In OpenShift/Kubernetes, an instance is called "pod"

Listing 1: Pod list returned by the API (excerpt)

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/namespaces/tomcat-in-the-cloud/pods",
    "resourceVersion": "7602"
  },
  "items": [
    {
      "metadata": {
        "name": "tomcat-in-the-cloud-1-5xbwm",
        "generateName": "tomcat-in-the-cloud-1-",
        "namespace": "tomcat-in-the-cloud",
        ...
        "creationTimestamp": "2017-06-17T13:36:10Z",
        ...
      },
      ...
      "status": {
        ...
        "hostIP": "192.168.42.74",
        "podIP": "172.17.0.3",
        "startTime": "2017-06-17T13:36:10Z",
        ...
      }
    },
    ...
  ]
}
```

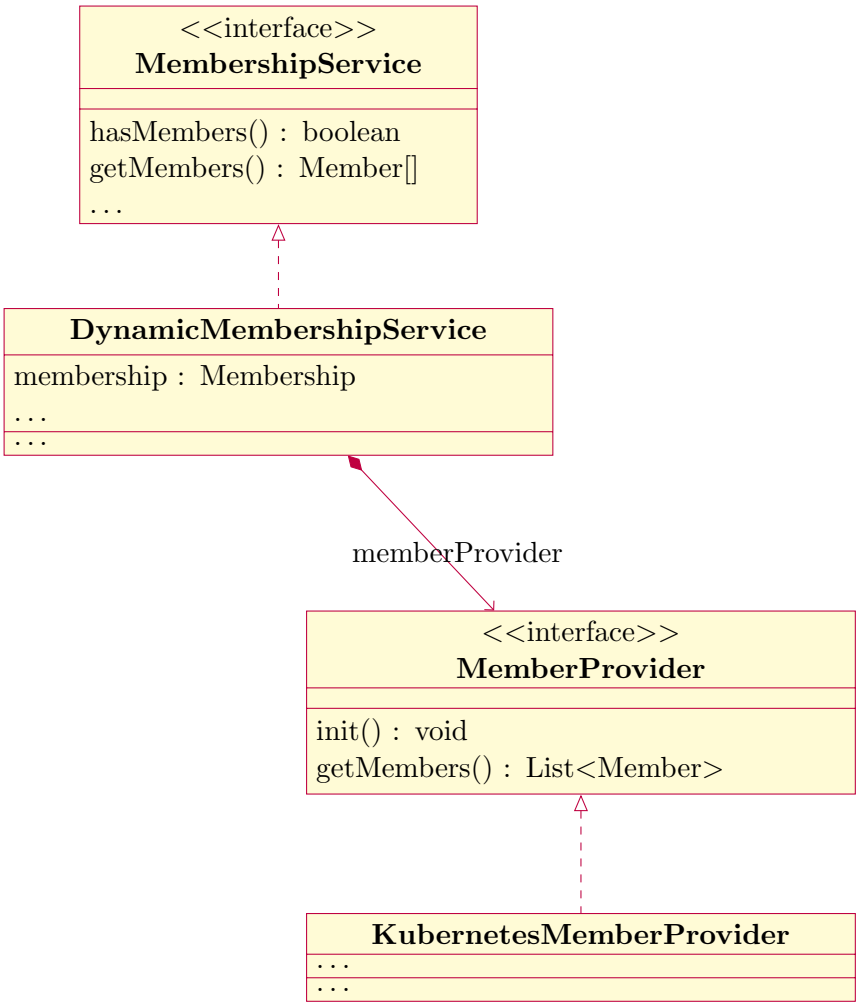

Figure 1: Principal classes and interfaces in our implementation.

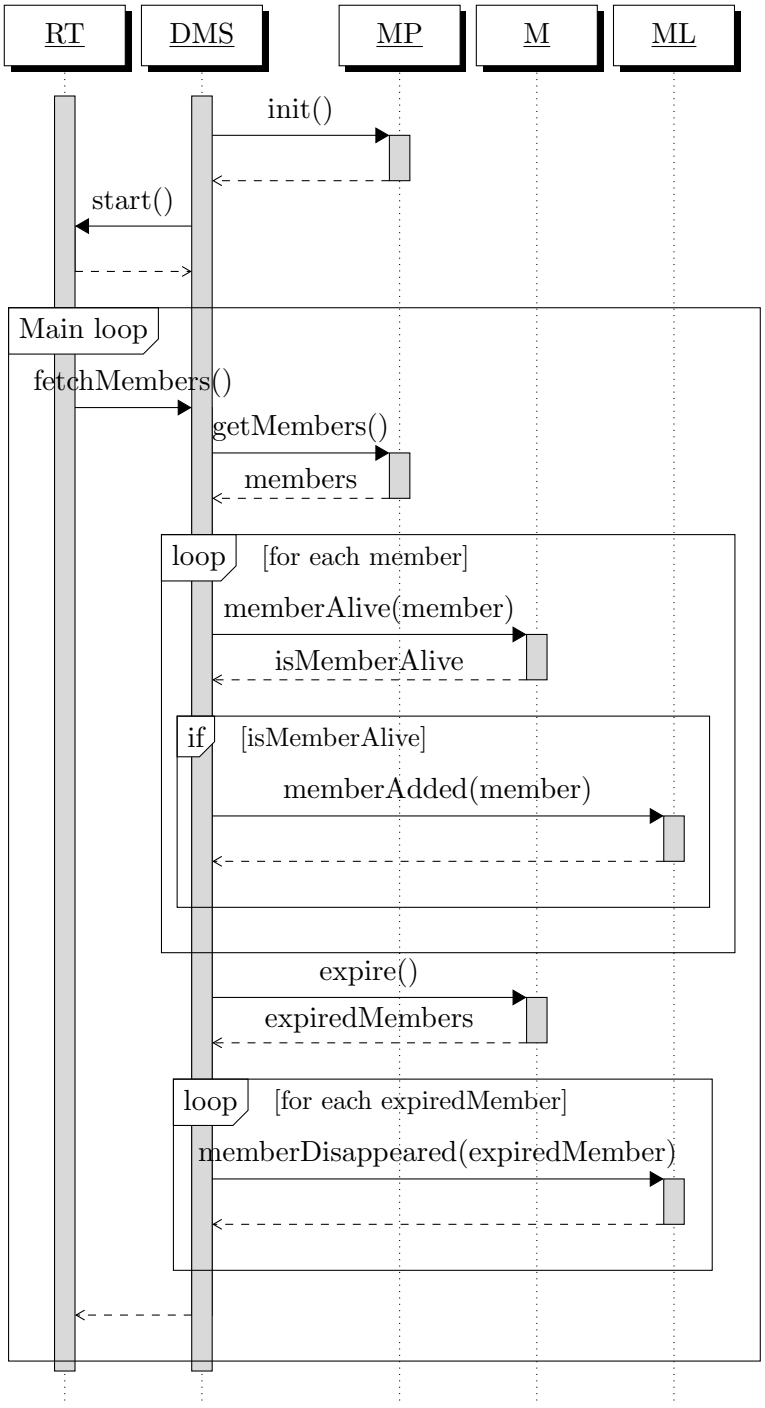

Figure 2: Sequence diagram for `DynamicMembershipService` (DMS). The other actors are `RefreshThread` (RT), `MembershipProvider` (MP), `Membership` (M) and `MembershipListener` (ML)

By making multiple requests and checking if the counter is incremented, we can verify if sessions are correctly replicated regardless of the Tomcat instance that replied. In between requests, we scaled the number of instances up and down by running

```
oc scale dc tomcat-in-the-cloud --replicas=$N
```

Where `$N` is the desired number of instances. This allowed to verify that our solution was robust to churn (members joining and leaving the cluster).

The solution was tested successfully in two different OpenShift deployments:

- On a 1-node OpenShift cluster, deployed by running `oc cluster up`
- On MiniShift[7], an all-in-one solution for deploying a local OpenShift instance for testing purposes.

Unfortunately, we didn't have time to test our solution on a multi-node cluster, but we are confident that it would work just as well.

6 Issues and future improvements

Although the solution is working as expected, some minor issues still exist.

- Error handling is at a bare minimum and should be improved.
- The class `CertificateStreamProvider`[5] isn't currently provided, as it wasn't needed in our test deployments and we wanted to avoid the additional `net.oauth` dependency.
- Although the class is named `KubernetesMemberProvider`, it might not work in non-OpenShift Kubernetes installations, since the environment variables used to set up API calls might be different. Testing in such environments is required.
- Multiple applications deployed to the same namespace/project will form a single cluster and share session data together, which adds some overhead (due to more data being shared) and can pose security risks. A solution to this could be possible with labels: `KubernetesMemberProvider` already filters pods by label if the environment variable `OPENSHIFT_KUBE_PING_LABELS` is set (can be set in the file *pom.xml*, similarly to `OPENSHIFT_KUBE_PING_LABELS`), and labels can be added the application by fabric8[6]
- Our testing method consisted of manually making requests and verifying that the counter was correctly incremented and that all the nodes were

[5] `https://github.com/jgroups-extras/jgroups-kubernetes/blob/master/src/main/java/org/jgroups/protocols/kubernetes/stream/CertificateStreamProvider.java`

[6] `https://maven.fabric8.io/#resource-labels-annotations`

hit at least once. Instead, a script could be written to perform this task automatically. This script could even be extended to automatically scale the number of pods. Such a setup would help iterating faster.

A Usage instruction

Create a new OpenShift project or switch to an existing one:

```
oc project tomcat-in-the-cloud
```

Authorize pods to see their peers:

```
oc policy add-role-to-user view \
    system:serviceaccount:$(oc project -q):default \
    -n $(oc project -q)
```

Deploy your application:

```
mvn fabric8:deploy
```

Customization

Project name

By default, the application is deployed to the project *tomcat-in-the-cloud*. To change this, modify the value of `fabric8.namespace` in *pom.xml*.

Server port

Tomcat is set to listen to port 8080, and fabric8 must expose this port in the resulting containers. This value is specified in two places: in the application itself (see *Main.java*), and in fabric8's configuration (variable `port` in *pom.xml*)

Number of replicas

Upon deployment, the application is automatically scaled to 2 pods. This value can be changed in *src/main/fabric8/deployment.yml*. When the default behavior (1 pod) is desired, this file can safely be deleted.

Sticky sessions

To make testing easier, sticky sessions have been disabled in *src/main/fabric8/route.yml*. To re-enable them, change `haproxy.router.openshift.io/disable_cookies` to `true` or simply delete the file.

When making changes to the route configuration, run mvn fabric8:undeploy before deploying again to make sure that all changes are taken into account.

References

[1] *Red Hat*. Red Hat. URL: http://www.redhat.com/en/about (visited on 2017-06-17).

[2] *Tomcat*. Apache Foundation. URL: http://tomcat.apache.org/ (visited on 2017-06-17).

[3] *OpenShift Container Platform*. Red Hat. URL: https://www.openshift.com/container-platform/index.html (visited on 2017-06-17).

[4] *Spring Framework*. Pivotal Software. URL: https://spring.io/ (visited on 2017-06-17).

[5] *KubePing*. JGroups. URL: https://github.com/jgroups-extras/jgroups-kubernetes (visited on 2017-06-17).

[6] *OpenShift REST API reference*. Red Hat. URL: https://docs.openshift.com/enterprise/3.0/rest_api/index.html (visited on 2017-06-17).

[7] *OpenShift Container Platform*. Red Hat. URL: https://www.openshift.org/minishift/ (visited on 2017-06-17).