Restcomm JAIN SLEE Clustering

Table of Contents

JAIN SLEE supports clustering, whether it is simple high availability () or complete fault tolerance () support. This is achieved through the replication of the container state. The Restcomm JAIN SLEE implementation also exposes a clustering extension for Resource Adaptors components, which live outside the container.

High Availability and Fault Tolerance

The used JAIN SLEE clustering mode is defined by the selected server profile:

JAIN SLEE reuses the JBoss Application Server clustering framework, and if all nodes of a cluster are in the same network then the underlying JBoss Application Server clustering will automatically handle the discovery of new cluster nodes and join these to the cluster. For more complicated setups, refer to the JBoss Application Server clustering documentation.

High Availability Mode

High availability mode provides no clustering functionality per say. The mode is useful when deploying for example single node, non-replicated or hot-cold configurations. In this mode all clustering needs to be explicitly done by the developer where applicable. In this sense, mode is not a clustered mode.

The `default` server profile is used to start the server in mode.

Fault Tolerant Mode

The fault tolerant mode is a fully clustered mode with state replication. An FT cluster can be viewed as one virtual container that extend over all the JAIN SLEE nodes that are active in the cluster. All activity context and Sbb entity data is replicated across the cluster nodes and is hence fully redundant. Events are not failed over, due to performance constraints, which means that an event fired and not yet routed will be lost if its cluster node fails.

The `all` server profile is used to start the server in mode.

Component Redundancy in Fault Tolerant Clusters

The fault tolerant clustering mode provides clustering for most of the JAIN SLEE components. JAIN SLEE components can be divided into internal and external components. Internal components are logically contained by the JAIN SLEE container, and external components are at least partly outside the container.

For a concrete example of how the container behaves in mode, see

[_fault_tolerant_cluster_example].

Internal Component Redundancy

Internal SLEE components are components that are completely inside the JAIN SLEE container. This group of components include entities, internal activities, events and timers. With the exception of events, all internal components will be fully redundant in a JAIN SLEE configuration.

SBB entities are fully replicated. entities are always serialized and saved by the container, regardless of the clustering profile. In an environment the container will replicate this serialized state to other nodes in the cluster so that it can be retrieved if the node fails or if the entity is processed in another node. All entities will hence be accessible by any node in the cluster at any given time.

Timers are fully replicated. Timers created in a given node will be executed in that same node. If the node leaves the cluster, all active timers from that node are recreated and run in another node.

Activity context interfaces (), as well as activity handles are fully replicated. The s for all activities are replicated within a fault tolerant cluster. However, the activity object is not replicated by default and needs to be handled by the resource adaptor that owns the activity in question if replication is required. The activity objects for all internal activities, e.g. null activities, profile table activities and service activities, are fully replicated.

Events are not replicated because of performance constraints. Hence, all events fired in a node is routed only in that node. However, if an event is fired in one node, and an entity created in another node has attached to that , the entity will be retrieved in the node that fired the event and the event will be delivered to it. Hence, even though the event is fired in a single node, the effects will be cluster-wide. Because the events are not replicated, any event currently being routed in a node that fails, will be lost.

External Component Redundancy

External JAIN SLEE components are components that are on the border between the SLEE container and the outside environment. This group of components include resource adaptors and external activities, neither of which are replicated by default.

The Resource adaptors may use the Fault Tolerant Resource Adaptor API extension of the JAIN SLEE 1.1 specification in order to be cluster-aware. Refer to the Fault Tolerant Resource Adaptor API and Resource Adaptor Activity Replication sections for more information on how to achieve resource adaptor and activity object redundancy.

Managing Components in Restcomm JAIN SLEE Cluster

JAIN SLEE clustering does not require special components management. Components can be deployed and undeployed in all cluster modes, including fault tolerant setups, and the cluster will handle the operation correctly. However, there are certain behaviours in fault tolerance setups to

be aware of:

`Service Activation`
: JAIN SLEE Service started events are only fired on the first cluster node started.

`Service Deactivation`
: Only the last node will force the removal of the service's entities.

`Resource Adaptor Entity Deactivation`
: Only the last node will force the removal of all its activities.

Fault Tolerant Resource Adaptor API

JAIN SLEE Resource Adaptors exist on the boundary between the container and the underlying protocol. The specification contract requires the object to implement the `javax.slee.resource.ResourceAdaptor`interface. This interface defines callbacks, which SLEE uses to interact with the , including one to provide the `javax.slee.resource.ResourceAdaptorContext`object. The Resource Adaptor Context provides object facilities to interact with SLEE.

The JAIN SLEE 1.1 RA API is a major milestone, standardizing RA and JSLEE contract. However, it misses an API for clustering, which is critical for a RA deployed in a clustered JAIN SLEE environment. The JAIN SLEE 1.1 contract does not define any fault tolerant data source nor cluster state callbacks.

The `Fault Tolerant RA API`extends the JAIN SLEE 1.1 RA , providing missing features related to clustering. An effort has been made keep the API similar to the standard RA contract, so that anyone who has developed a JAIN SLEE 1.1 RA is able to easily use the proprietary API extension.

The Fault Tolerant Resource Adaptor Object

The core of the Fault Tolerant RA API is the `org.mobicents.slee.resource.cluster.FaultTolerantResourceAdaptor` `interface. It is intended to be used instead of the` `javax.slee.resource.ResourceAdaptor`interface from the JAIN SLEE 1.1 Specification.

The FaultTolerant interface provides three new callback methods used by the container:

`setFaultTolerantResourceAdaptorContext(FaultTolerantResourceAdaptorContext context)`
: This method provides the RA with the `org.mobicents.slee.resource.cluster.FaultTolerantResourceAdaptorContext` `object, which gives access to facilities related with the cluster. This method is invoked by SLEE after invoking` `raConfigure(ConfigProperties properties)`from JAIN SLEE 1.1 specs.

`unsetFaultTolerantResourceAdaptorContext()`
: This method indicates that the RA should remove any references it has to the FaultTolerantResourceAdaptorContext, as it is not valid anymore. The method is invoked by SLEE before invoking `unsetResourceAdaptorContext()`from JAIN SLEE 1.1 specs.

`failOver(K key)`
: Callback from SLEE when the local RA was selected to recover the state for a replicated data key, which was owned by a cluster member that failed. The RA may then restore any runtime resources associated with such data.

`dataRemoved(K key)`
: Optional callback from SLEE when the replicated data key was removed from the cluster, this may be helpful when the local RA maintains local state.

The Fault Tolerant Resource Adaptor Context

The clustered RA context follows the contract of JAIN SLEE 1.1 specification interface `javax.slee.resource.ResourceAdaptorContext`. It gives access to facilities that the RA may use when run in a clustered environment.

The cluster contract is defined in: `org.mobicents.slee.resource.cluster.FaultTolerantResourceAdaptorContext`. It provides critical information, such as if SLEE is running in local mode (not clustered), if it is the head/master member of the cluster, and what the members of the cluster are.

The Fault Tolerant Resource Adaptor Replicated Data Sources

The Fault Tolerant Resource Adaptor Context provides two data sources to replicate data in cluster:

`ReplicatedData`
: A container for serializable data, which is replicated in the SLEE cluster, but don't require any failover.

`ReplicatedDataWithFailover`
: A `ReplicatedData`which requires fail over callbacks, this means, that for all data stored here, when a cluster member goes down, the SLEE in another cluster member will invoke the `failOver(Key k)`callback in the Fault Tolerant RA object.

When retrieved from the context through a boolean parameter, both types of `ReplicatedData` can activate the callback on the `FaultTolerantResourceAdaptor`which indicates that a specific data was removed from the cluster remotely.

The Fault Tolerant Resource Adaptor Timer

The standard Resource Adaptor Context provides a `java.util.Timer`, which can be used by the Resource Adaptor to schedule the execution of tasks, the Fault Tolerant Resource Adaptor Context provides `org.mobicents.slee.resource.cluster.FaultTolerantTimer`, an alternative scheduler which is able to fail over tasks scheduled.

The Fault Tolerant Timer has an interface that resembles the JDK's `ScheduledExecutorService`, with two fundamental changes to allow a proper interaction in a cluster environment:

`Task Interface`
: Instead of relying on pure `Runnable` tasks, tasks must follow a specific interface

`FaultTolerantTimerTask`, to ensure that the timer is able to replicate the task's data, and failover the task in any cluster node.

`Task Cancellation`
: Cancellation of task is done through the Timer interface, not through `ScheduledFuture` objects, this allows the operation to be easily done in any cluster node.

The Fault Tolerant Timer interface:

`cancel(Serializable taskID)`
: Requests the cancellation of the FT Timer Task with the specified ID.

`configure(FaultTolerantTimerTaskFactory taskFactory, int threads)`
: Configures the fault tolerant timer, specifying the timer task factory and the number of threads the timer uses to execute tasks.

`isConfigured()`
: Indicates if the timer is configured.

`schedule(FaultTolerantTimerTask task, long delay, TimeUnit unit)`
: Creates and executes a one-shot action that becomes enabled after the given delay.

`scheduleAtFixedRate(FaultTolerantTimerTask task, long initialDelay, long period, TimeUnit unit)`:: Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is executions will commence after `initialDelay` then `initialDelay+period`, then `initialDelay + 2 * period`, and so on. If any execution of the task encounters an exception, subsequent executions are suppressed. Otherwise, the task will only terminate via cancellation or termination of the executor. If any execution of this task takes longer than its period, then subsequent executions may start late, but will not concurrently execute.

`scheduleWithFixedDelay(FaultTolerantTimerTask task, long initialDelay, long delay, TimeUnit unit)`:: Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given delay between the termination of one execution and the commencement of the next. If any execution of the task encounters an exception, subsequent executions are suppressed. Otherwise, the task will only terminate via cancellation or termination of the executor.

There is a single Fault Tolerant Timer per RA Entity, and when first retrieved, and before any task can be scheduled, the Fault Tolerant Timer must be configured, through its configure(...) method.

The Fault Tolerant Resource Adaptor Timer Task

As mentioned in previous section, tasks submitted to the Fault Tolerant Timer must follow a specific interface, `FaulTolerantTimerTask`, it is nothing more than a `Runnable`, which provides the replicable `FaultTolerantTimerTaskData`. The task data must be serializable and provide a Serializable task ID, which identifies the task, and may be used to cancel its execution.

The Fault Tolerant Resource Adaptor Timer Example Usage

A simple example for the usage of the Faul Tolerant Timer Task:

```
// data, task and factory implementation

package org.mobicents.slee.resource.sip11;

import java.io.Serializable;

import org.mobicents.slee.resource.cluster.FaultTolerantTimerTaskData;

public class FaultTolerantTimerTaskDataImpl implements
        FaultTolerantTimerTaskData {

    private final String taskID;

    public FaultTolerantTimerTaskDataImpl(String taskID) {
        this.taskID = taskID;
    }

    @Override
    public Serializable getTaskID() {
        return taskID;
    }
}

package org.mobicents.slee.resource.sip11;

import org.mobicents.slee.resource.cluster.FaultTolerantTimerTask;
import org.mobicents.slee.resource.cluster.FaultTolerantTimerTaskData;
import org.mobicents.slee.resource.cluster.FaultTolerantTimerTaskFactory;

public class FaultTolerantTimerTaskFactoryImpl implements
        FaultTolerantTimerTaskFactory {

    private final SipResourceAdaptor ra;

    public FaultTolerantTimerTaskFactoryImpl(SipResourceAdaptor ra) {
        this.ra = ra;
    }

    @Override
    public FaultTolerantTimerTask getTask(FaultTolerantTimerTaskData data) {
        return new FaultTolerantTimerTaskImpl(ra, data);
    }
}

package org.mobicents.slee.resource.sip11;

import org.mobicents.slee.resource.cluster.FaultTolerantTimerTask;
```

```
import org.mobicents.slee.resource.cluster.FaultTolerantTimerTaskData;

public class FaultTolerantTimerTaskImpl implements FaultTolerantTimerTask {

    private final SipResourceAdaptor ra;
    private final FaultTolerantTimerTaskData data;

    public FaultTolerantTimerTaskImpl(SipResourceAdaptor ra,
            FaultTolerantTimerTaskData data) {
        this.ra = ra;
        this.data = data;
    }

    @Override
    public void run() {
        ra.getTracer("FaultTolerantTimerTaskImpl").info("Timer executed.");
    }

    @Override
    public FaultTolerantTimerTaskData getTaskData() {
        return data;
    }
}

// ra code retrieving the timer, configuring it and submiting a task

public void setFaultTolerantResourceAdaptorContext(
        FaultTolerantResourceAdaptorContext<SipActivityHandle, String> context) {
    this.ftRaContext = context;
    FaultTolerantTimer timer = context.getFaultTolerantTimer();
    timer.config(new FaultTolerantTimerTaskFactoryImpl(this), 4);
    FaultTolerantTimerTaskDataImpl data = new FaultTolerantTimerTaskDataImpl("xyz");
    FaultTolerantTimerTaskImpl task = new FaultTolerantTimerTaskImpl(this,
            data);
    timer.schedule(task, 30, TimeUnit.SECONDS);
}
```

Resource Adaptor Activity Replication

The Resource Adaptor API includes an optional component named `javax.slee.resource.Marshaler`, which is responsible, besides other functions, for converting Activity Handles to byte arrays and vice-versa. Also relevant, the Resource Adaptor, when starting activities, may provide a flag indicating that the container may marshall the activity (using the Marshaler). In case of a container cluster with data replication, if an activity is to be replicated then the Marshaler must be provided and the activity flags must activate the flag MAY_MARSHALL, otherwise the activity is not replicated and if a node fails all its activities are removed from the container cluster.

The activity replication doesn't mean that the activity object is replicated by any means, only the related Activity Handle. The Resource Adaptor must use the Fault Tolerant RA API or its own means to replicate any additional data to support that presence of the activity in all nodes of the cluster. Usage of the Fault Tolerant RA API is recommended since it reuses the clustering setup of the container.