**This version of Akka (akka / 2.3.1) is outdated and not supported!**

Please upgrade to version [**2.5.3**](http://doc.akka.io/docs/akka/2.5.3/index.html) as soon as possible.

You are browsing the docs for Akka 2.3.1, however the latest release in this series is: [**2.3.16**](http://doc.akka.io/docs/akka/2.3.16/contrib/cluster-singleton.html).

Dismiss Warning for a Day

[http://doc.akka.io/docs/akka/2.3.1/_static/logo-small.png](http://akka.io/)

* [Documentation](http://akka.io/docs)
* [FAQ](http://akka.io/faq)
* [Download](http://akka.io/downloads)
* [Mailing List](http://groups.google.com/group/akka-user)
* [Code](http://github.com/akka/akka)
* [Commercial Support](http://www.typesafe.com/how/subscription)

Cluster Singleton

[](http://doc.akka.io/docs/akka/2.3.1/AkkaScala.pdf)

[](http://doc.akka.io/docs/akka/2.3.1/AkkaJava.pdf)

* | [Cluster Sharding](http://doc.akka.io/docs/akka/2.3.1/contrib/cluster-sharding.html) »
* [Contents](http://doc.akka.io/docs/akka/2.3.1/index.html)
* « [Mailbox with Explicit Acknowledgement](http://doc.akka.io/docs/akka/2.3.1/contrib/peek-mailbox.html) |
* **Version 2.3.1**

Top of Form

|  |  |  |
| --- | --- | --- |
|  |  |  |

Bottom of Form

For some use cases it is convenient and sometimes also mandatory to ensure that you have exactly one actor of a certain type running somewhere in the cluster.

Some examples:

* single point of responsibility for certain cluster-wide consistent decisions, or coordination of actions across the cluster system
* single entry point to an external system
* single master, many workers
* centralized naming service, or routing logic

Using a singleton should not be the first design choice. It has several drawbacks, such as single-point of bottleneck. Single-point of failure is also a relevant concern, but for some cases this feature takes care of that by making sure that another singleton instance will eventually be started.

The cluster singleton pattern is implemented by akka.contrib.pattern.ClusterSingletonManager. It manages one singleton actor instance among all cluster nodes or a group of nodes tagged with a specific role. ClusterSingletonManager is an actor that is supposed to be started on all nodes, or all nodes with specified role, in the cluster. The actual singleton actor is started by the ClusterSingletonManager on the oldest node by creating a child actor from supplied Props. ClusterSingletonManager makes sure that at most one singleton instance is running at any point in time.

The singleton actor is always running on the oldest member, which can be determined by Member#isOlderThan. This can change when removing that member from the cluster. Be aware that there is a short time period when there is no active singleton during the hand-over process.

The cluster failure detector will notice when oldest node becomes unreachable due to things like JVM crash, hard shut down, or network failure. Then a new oldest node will take over and a new singleton actor is created. For these failure scenarios there will not be a graceful hand-over, but more than one active singletons is prevented by all reasonable means. Some corner cases are eventually resolved by configurable timeouts.

You can access the singleton actor by using the provided akka.contrib.pattern.ClusterSingletonProxy, which will route all messages to the current instance of the singleton. The proxy will keep track of the oldest node in the cluster and resolve the singleton's ActorRef by explicitly sending the singleton's actorSelectionthe akka.actor.Identify message and waiting for it to reply. This is performed periodically if the singleton doesn't reply within a certain (configurable) time. Given the implementation, there might be periods of time during which the ActorRef is unavailable, e.g., when a node leaves the cluster. In these cases, the proxy will stash away all messages until it is able to identify the singleton. It's worth noting that messages can always be lost because of the distributed nature of these actors. As always, additional logic should be implemented in the singleton (acknowledgement) and in the client (retry) actors to ensure at-least-once message delivery.

**An Example**

Assume that we need one single entry point to an external system. An actor that receives messages from a JMS queue with the strict requirement that only one JMS consumer must exist to be make sure that the messages are processed in order. That is perhaps not how one would like to design things, but a typical real-world scenario when integrating with external systems.

On each node in the cluster you need to start the ClusterSingletonManager and supply the Props of the singleton actor, in this case the JMS queue consumer.

In Scala:

1. system.actorOf(ClusterSingletonManager.props(
2. singletonProps = Props(classOf[Consumer], queue, testActor),
3. singletonName = "consumer",
4. terminationMessage = End,
5. role = Some("worker")),
6. name = "singleton")

Here we limit the singleton to nodes tagged with the "worker" role, but all nodes, independent of role, can be used by specifying None as role parameter.

The corresponding Java API for the singeltonProps function is akka.contrib.pattern.ClusterSingletonPropsFactory. The Java API takes a plain String for the role parameter and null means that all nodes, independent of role, are used.

In Java:

1. system.actorOf(ClusterSingletonManager.defaultProps(Props.create(Consumer.class, queue, testActor), "consumer",
2. new End(), "worker"), "singleton");

**Note**

The singletonProps/singletonPropsFactory is invoked when creating the singleton actor and it must not use members that are not thread safe, e.g. mutable state in enclosing actor.

Here we use an application specific terminationMessage to be able to close the resources before actually stopping the singleton actor. Note that PoisonPill is a perfectly fine terminationMessage if you only need to stop the actor.

Here is how the singleton actor handles the terminationMessage in this example.

1. case End ⇒
2. queue ! UnregisterConsumer
3. case UnregistrationOk ⇒
4. context stop self
5. case Ping ⇒
6. sender ! Pong

Note that you can send back current state to the ClusterSingletonManager before terminating. This message will be sent over to the ClusterSingletonManager at the new oldest node and it will be passed to the singletonProps factory when creating the new singleton instance.

With the names given above, access to the singleton can be obtained from any cluster node using a properly configured proxy.

In Scala:

1. system.actorOf(ClusterSingletonProxy.props(
2. singletonPath = "/user/singleton/consumer",
3. role = Some("worker")),
4. name = "consumerProxy")

In Java:

1. system.actorOf(ClusterSingletonProxy.defaultProps("user/singleton/consumer", "worker"), "consumerProxy");

A more comprehensive sample is available in the [Typesafe Activator](http://www.typesafe.com/platform/getstarted) tutorial named [Distributed workers with Akka and Scala!](http://www.typesafe.com/activator/template/akka-distributed-workers) and [Distributed workers with Akka and Java!](http://www.typesafe.com/activator/template/akka-distributed-workers-java).

**Contents**

* [**An Example**](http://doc.akka.io/docs/akka/2.3.1/contrib/cluster-singleton.html#An_Example)
* **Akka**
* [Documentation](http://akka.io/docs)
* [FAQ](http://akka.io/faq)
* [Downloads](http://akka.io/downloads)
* [News](http://akka.io/news)
* [Blog](http://letitcrash.com/)
* **Contribute**
* [Community Projects](http://akka.io/community)
* [Source Code](http://github.com/akka/akka)
* [Mailing List](http://groups.google.com/group/akka-user)
* [Report a Bug](http://doc.akka.io/docs/akka/current/project/issue-tracking.html)
* **Company**
* [Commercial Support](http://www.typesafe.com/how/subscription)
* [Team](http://akka.io/team)
* [Contact](mailto:info@typesafe.com)
* 

© 2014 [Typesafe Inc.](http://www.typesafe.com/) Akka is Open Source and available under the Apache 2 License.

Last updated: Mar 25, 2014