Akka for Distributed Systems

Akka Artery Remoting and Akka Cluster: Introduction

Gianluca Aguzzi gianluca.aguzzi@unibo.it

Concurrent and Distributed Programming course
Department of Computer Science and Engineering (DISI)
textscAlma Mater Studiorum – Università di Bologna

16/05/2022

Contents

1 Akka Remote Artery

2 Akka Clustering

Artery Remoting 🔗 (supersedes Classic Remoting)

- **(Artery) Remoting** is the support by which **actor systems on different nodes** can talk to each other in a **peer-to-peer** fashion.
- **Location transparency** 🔗. No API difference between local or remote systems
  - `ActorRefs` to remote actors look like those to local actors
- **Serialization**. For interaction across a network, messages must be **de/serialisable**

❗ No meant to be used directly!!

Use higher-level modules like **Akka Cluster** utilities or **technology-agnostic protocols** such as HTTP and gRPC (cf. *Akka HTTP* and *Akka gRPC*—which implement HTTP/gRPC stack on top of `akka-actor` and `akka-stream`)

Configuration

application.conf

```
akka {
  actor {
    provider = remote // local or remote or cluster
    serialization-bindings {
      "it.unibo.pcd.akka.Message" = jackson-cbor // serialization binding, in next slides
    }
  }
  remote { // remote configuration
    artery {
      transport = tcp # aeron-udp, tls-tcp
      canonical.hostname = "127.0.0.1" // in real deployments,
      canonical.port = 25520
    }
  }
}
```

Acquiring references to remote actors

- You can use remote `ActorRef` as local one (i.e, `ref ! msg`) ...
- ! ...But you need to **obtain** an `ActorRef`!!
- Two potential ways:
  - creating a (remote) actor: supported through akka classic with `actorSelection` (i.e. retrieve an `ActorRef` from an URL)
  - passing an `ActorRef` in a message
- **Akka typed**: `Receptionist` could be used for registring & finding `ActorRef`

Serialization

- In order to send message to remote peers, you should devise your serialization policy.

1. You need to enable **serialization for messages** (automatic when using remoting/cluster)
2. Choice of serialization mechanism (Recommended: **Jackson**)

application.conf

```
akka {
  actor {
    provider = remote // local or remote or cluster
    serializers { // key value map to associate name to serializers
      // Defaults..
      jackson-json = "akka.serialization.jackson.JacksonJsonSerializer"
      jackson-cbor = "akka.serialization.jackson.JacksonCborSerializer"
      proto = "akka.remote.serialization.ProtobufSerializer"
      // Ad-hoc serializers
      myown = "docs.serialization.MyOwnSerializer"
    },
    serialization-bindings { // key value map to link root message interface to serializer
      "it.unibo.pcd.akka.Message" = jackson-cbor // serialization binding, in next slides
    }
  } ...
}
```

Include the dependency on your serialisers.

```
libraryDependencies +=
  "com.typesafe.akka" %% "akka-serialization-jackson" % akkaVersion
```

Delivery guarantees, remote watch, and quarantine

- **Akka guarantees**: (1) **at-most-once delivery**; and (2) **message ordering between pairs of actors**
- Akka Remoting uses **TCP** or **Aeron** (which adds reliable delivery and session semantics on UDP) as "reliable" underlying message transport
- Cases when messages may not be delivered to destination
  - during a **network partition** (TCP connection / Aeron session broke)
  - when **sending too many messages without flow** control filling up the *outbound send queue*
  - on **de/serialization failure**
  - **exception in the remoting infrastructure**
- **Remote watch**: you can watch remote actors just like local actors
  - A *failure detector* uses hearth beats to detect failures and generate `Terminated`
- **System messages** for remote death watch are delivered with "exactly once" guarantee
  - If a system message cannot be delivered the association with the destination system is irrecoverably/failed, and `Terminated` is signaled to local actors for all watched actors on the remote system. The destination system enters in the *quarantined state*.
  - The only way to recover from quarantine is to **restart** the actor system.

Contents

1 Akka Remote Artery

2 Akka Clustering

Akka Cluster Specification (Typed) 🔗 (1/2)

- **Overview**: Akka Cluster provides a fault-tolerant decentralized peer-to-peer based *Cluster Membership Service* with no single point of failure or single point of bottleneck. It does this using *gossip* protocols and an automatic *failure detector*.
- **Motivation**: Akka Cluster allows for building distributed applications, where one application or service spans multiple nodes (in practice **multiple `ActorSystems`**).

Concepts

- **node**: **logical member of cluster, identified by hostname:port:uid** (there could be multiple nodes on the same physical machine)
- **cluster**: set of nodes joined together through *Cluster Membership Service*
- **leader**: cluster node that manages cluster convergence and membership state transitions

Akka Cluster Specification (Typed) 🔗 (2/2)

Cluster membership: how it works ➔ gossip

- **Vector clocks** used to reconcile and merge differences in cluster state during gossiping.
- **Convergence**: when all nodes are in the *seen set* for current cluster state
- Note: convergence cannot occur when some node is **unreachable** (cf. *split brain*)
- A *Split brain resolver* deals with partitions; can be configured with downing strategies
- A *failure detector* is what tries to **detect** if a node is **un/reachable**

Akka Cluster (Typed): basic usage (1/2)

build.sbt

```
val AkkaVersion = "2.6.19"
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-cluster-typed" % AkkaVersion,
  "com.typesafe.akka" %% "akka-serialization-jackson" % akkaVersion)
```

- The Cluster extension gives you access to **management tasks** such as `Joining`, `Leaving` and `Downing` and subscription of cluster membership events such as `MemberUp`, `MemberRemoved` and `UnreachableMember`, which are exposed as event APIs.

  Access the Cluster extension on a node

  ```
  val cluster = Cluster(system)
  ```

- Key reference on the Cluster extension:
  - **manager**: an **ActorRef[ClusterCommand]** where a `ClusterCommand` is a command such as: `Join`, `Leave` (graceful exit) and `Down` (node has crashed)
  - **subscriptions**: an `ActorRef[ClusterStateSubscription]`
  - **state**: the current `CurrentClusterState`

Akka Cluster (Typed): basic usage (2/2)

Access the **Cluster** extension on a node

```
akka {
  actor {
    provider = "cluster"
    serialization-bindings {
      "it.unibo.pcd.akka.Message" = jackson-cbor
    }
  }
  remote.artery {
    canonical {
      hostname = "127.0.0.1"
      port = 2551
    }
  }

  cluster {
    seed-nodes = [
      "akka://ClusterSystem@127.0.0.1:2551",
      "akka://ClusterSystem@127.0.0.1:2552"]

    downing-provider-class = "akka.cluster.sbr.SplitBrainResolverProvider"
  }
}
```

- Official Examples: `https://github.com/akka/akka-samples` (adapted in `https://github.com/cric96/pcd-lab-akka-distributed`)

Cluster Membership (1/2)

Joining

- **Joining a cluster programmatically** (without using seed nodes)

```
cluster.manager ! Join(cluster.selfMember.address)
```

- **Joining through seed nodes** (point of contact for new nodes that join to the cluster)

1. Join automatically to seed nodes with **Cluster Bootstrap**
2. Join configured seed nodes:
   `akka.cluster.seed-nodes=["akka://Sys@host1:2552",..]`
   - The **first seed must be started first** to allow other seeds to join
3. Join seed nodes programmatically

```
val seedNodes: List[Address] = // discover in some way
Cluster(system).manager ! JoinSeedNodes(seedNodes)
```

Cluster Membership (2/2)

Leaving a cluster

- **Leaving a cluster programmatically** (similar to *downing* a node)

```
cluster2.manager ! Leave(cluster2.selfMember.address)
```

- **Graceful exit via Coordinated Shutdown**: e.g. `sys.terminate()` or by root actor termination
- **Graceful exit via HTTP or JMX**
- **Non-graceful exit**. E.g. in case of abrupt termination, the node will be detected as **unreachable** by other nodes and removed after `Downing`.

Subscriptions

- **Receive cluster state changes** (**subscriptions**) 🔗: e.g. to be notified of a node leaving the cluster

```
val subscriber: ActorRef[MemberEvent]
cluster.subscriptions ! Subscribe(subscriber, classOf[MemberEvent])
```

Node roles

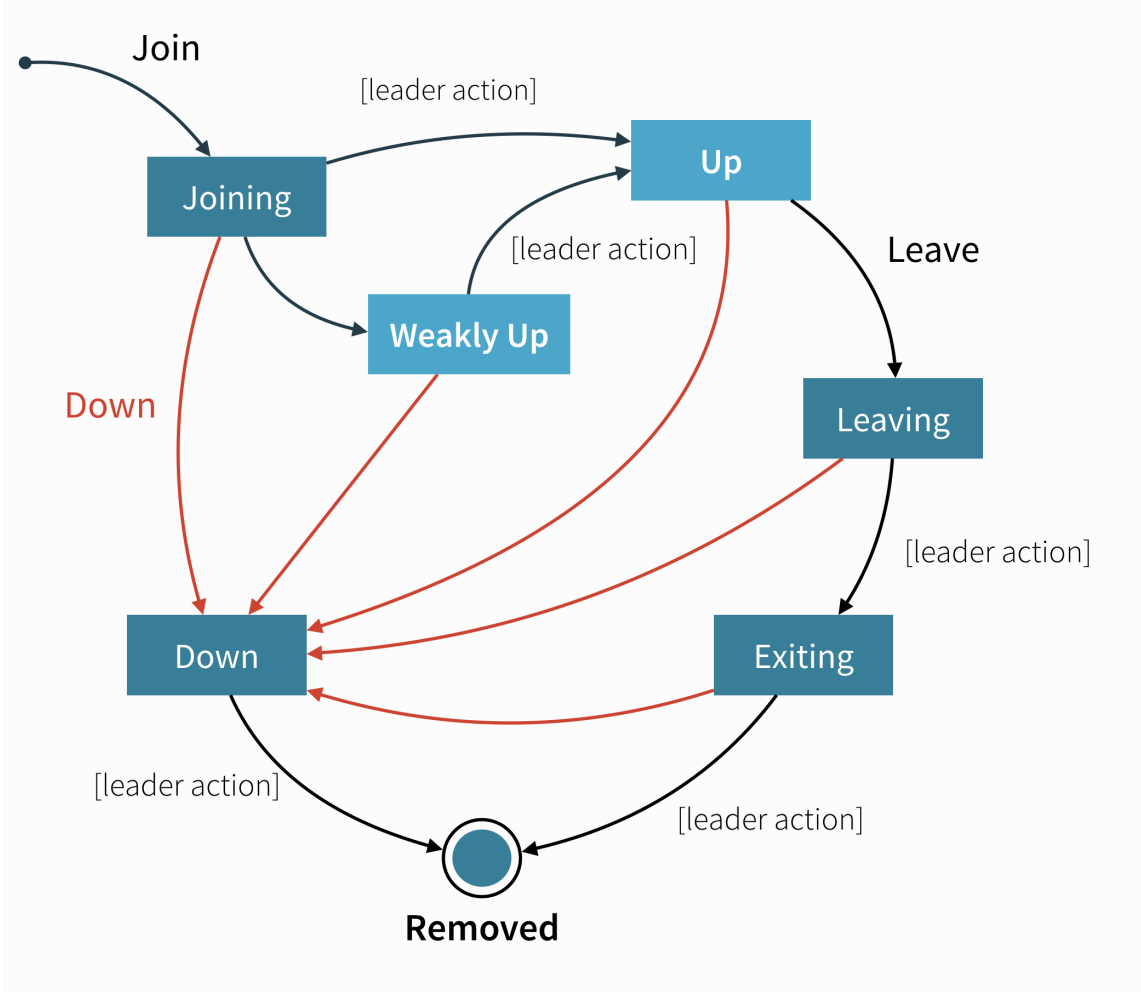
- **Motivation**. Not all nodes of a cluster need to perform the same function. **Choosing which actors to start on each node can take roles into account to properly distribute responsibilities**
- Config key **akka.cluster.roles**
- **Getting role info.**
  - Role info is included in membership information (cf. `MemberEvent`)
  - For the own node, `cluster.selfMember.hasRole(r)`

Access the Cluster extension on a node

```
val selfMember = Cluster(context.system).selfMember
if (selfMember.hasRole("backend")) {
  context.spawn(Backend(), "back")
} else ...
```

Cluster membership

- **Cluster membership**: service keeping track what nodes are **members** and their **health**
- A **leader** confirms state changes when **convergence** on membership state is reached



Akka Cluster facilities

- **Receptionist** 🔗. Registered actors will appear in the receptionist of other nodes of the cluster.
  - The state for the receptionist is propagated via `distributed-data` support
- **Group router** 🔗. Created for a `ServiceKey`, uses receptionist to discover actors, and routes messages to available actors
  - By using the receptionist, this is cluster-aware out-of-the-box
- **Distributed data** 🔗. The `DistributedData` extension provides a cluster-wide **key-value** store where values are CRDTs
  - Local updates + replication via gossip + conflict-resolution via monotonic merge function
  - Data types include counters, sets, maps, registers
  - Read/write access via `DistributedData(ctx.system).replicator`
- Cluster singleton 🔗. Support for managing one singleton actor in the cluster

```
ClusterSingleton(system).init(SingletonActor(someBehavior()), "MySingleton")
```

- Cluster sharding 🔗 🔗 🔗. Distributed and interact with actors based on their logical ID.
  - A **Shard** is a group of **entities** managed together.
  - Each cluster node has a **ShardRegion** actor that extracts the entity and shard IDs from incoming messages
  - A singleton **ShardCoordinator** decides which `ShardRegion` shall own what `Shards`; the state of shard locations is shared via `distributed-data`

Acknowledgement

The material and the slides are derived from Roberto Casadei works 🔗

Akka for Distributed Systems

Akka Artery Remoting and Akka Cluster: Introduction

Gianluca Aguzzi gianluca.aguzzi@unibo.it

Concurrent and Distributed Programming course
Department of Computer Science and Engineering (DISI)
textscAlma Mater Studiorum – Università di Bologna

16/05/2022

References

[] *Documentation | Akka*. `https://akka.io/docs/`. (Accessed on 05/07/2022).