Pragmukko

Pragmukko

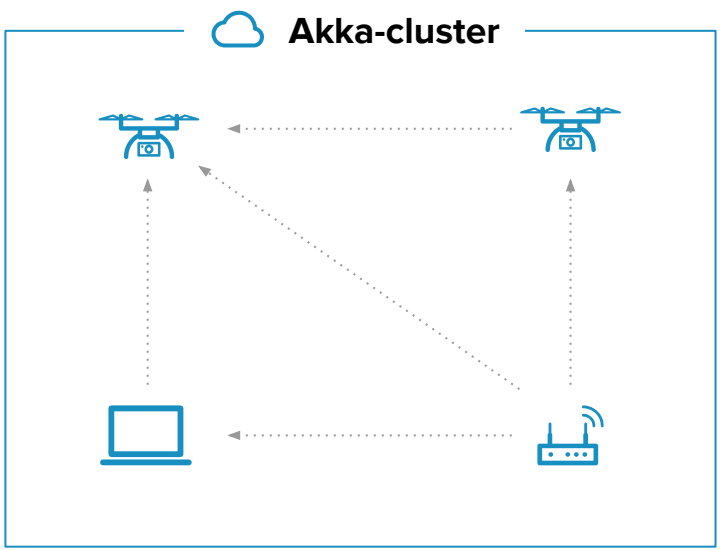
- Pragmukko is an Akka-based framework for distributed computing.
- Originally, it was developed for building an IoT solution but it also fit well for many other cases.
- Pragmukko treats an IoT solution as a cluster. Each device in such solution is running under a cluster node.
- From Akka, Pragmukko inherits the actor model and clustering features.

Goals

Pragmukko features:

- Communication
- Clustering
- Integration with 3rd party systems
- Interaction with hardware (due to its IoT origin)
- Fault tolerance
- Service discovering

The Idea



Pragmukko treats the IoT cloud as an Akka-cluster.
Each device runs an Akka-node, which in Pragmukko terms is called Pragma.

There are several actors within a Pragma. Some of them handle system messages, other run user-defined logic.

Also, Akka provides **Pragmukko** with cluster features, so that each node can be added or removed dynamically, as well as each node obtains information about cluster state.

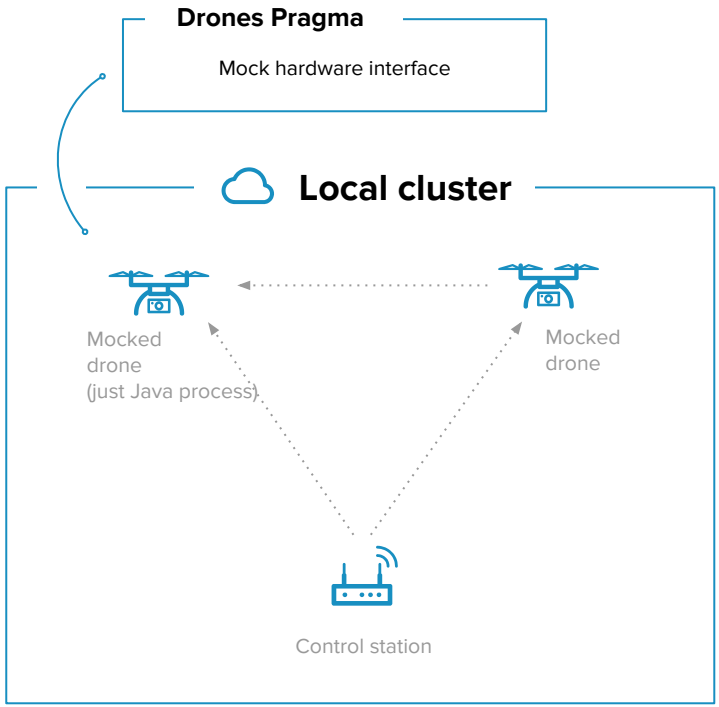
Platform for IoT Solutions

Pragmukko was developed to design IoT solutions.

Pragmukko provides the layer for interaction with **various hardware**. Also, it is extensible and easily adds support of new types of devices.

One of the goals of Pragmukko is **uniform** development for different types of devices.

Feature: IoT Simulation



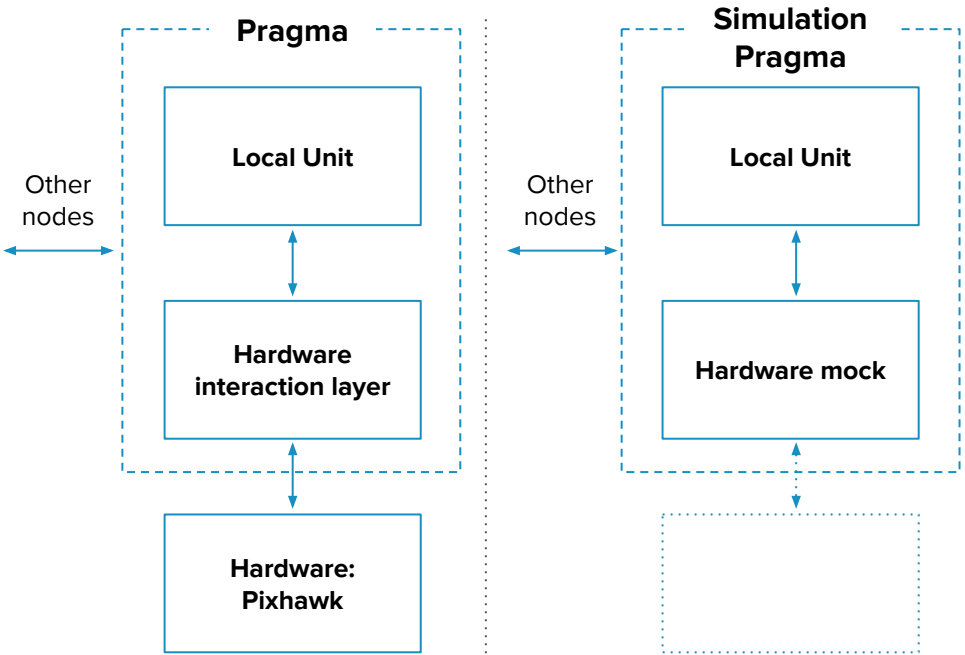
Another approach for IoT development is device simulation.

It is possible to run simulating Pragmas within one OS as simple Java processes, and they will behave as real devices – send telemetry, react on commands, and more.

So, whenever needed, you can deploy a simulating IoT cloud for testing, debugging, or experimenting, even without devices.

IoT Integration Test? – Sounds weird, but why not.

Feature: IoT Simulation



The simulating Pragmas don't interact with real hardware.

Instead, they use a so-called mock hardware interface that simulates real hardware.

Therefore, it becomes possible to run even a number of Pragmas within one process.

There are several predefined mock hardware interfaces. Also, it's easy to create a new one.

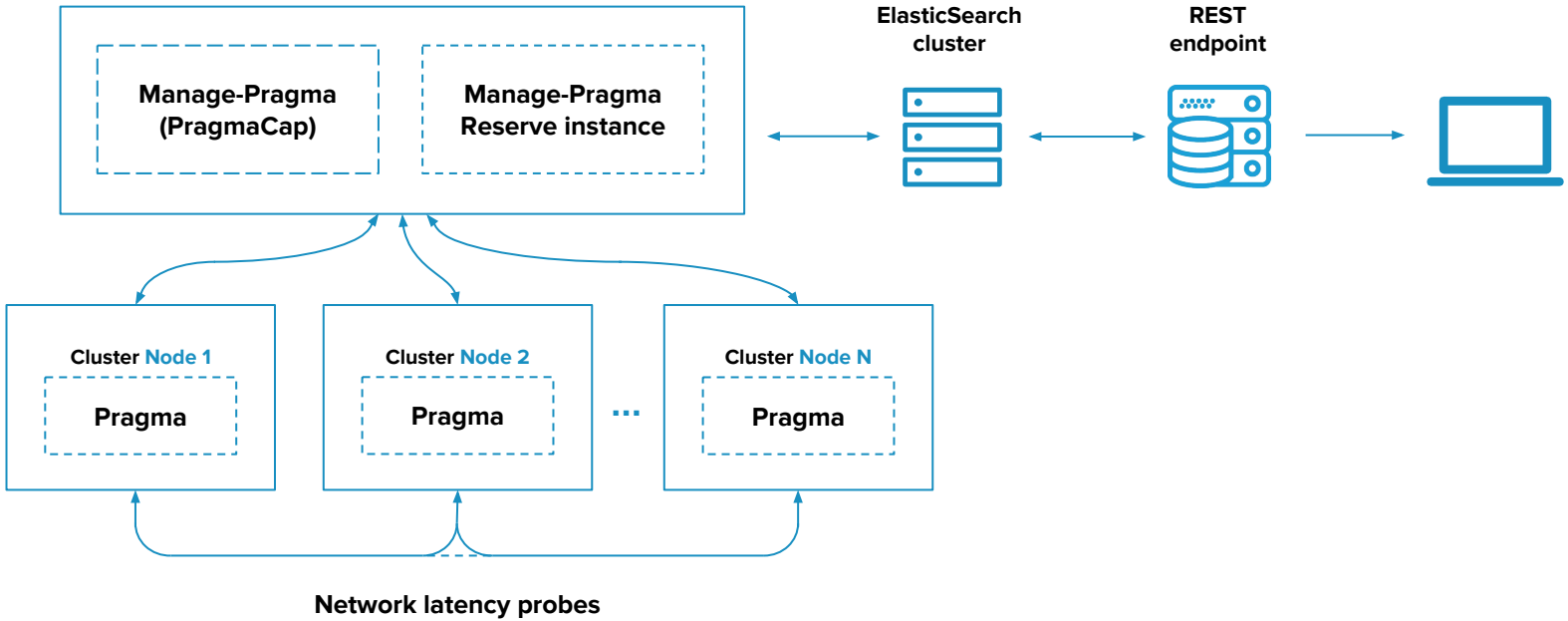
Pragnaky

Another option of Pragmukko usage is development of distributed applications.

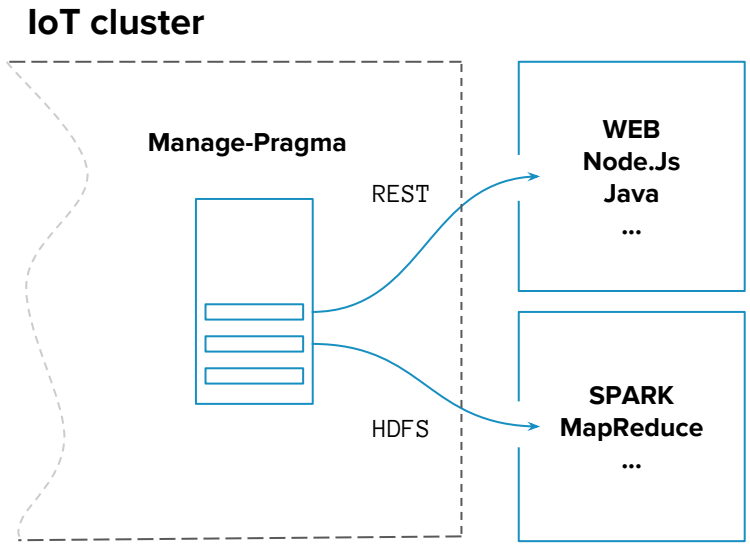
Pragnaky is a cluster health monitoring tool, a component of **Mantl**, that was created using Pragmukko.

Pragnaky includes extended Pragmas on each physical cluster node for obtaining telemetry and measuring network latencies between hosts, and special Pragmas for collecting telemetry from all pragmas and storing it to ElasticSearch.

Pragnaky Schema



Integration with the 3rd Party Solutions



New features can be added to Pragmas by extensions. A Pragma extension is an actor which can be subscribed on different messages from an Akka cluster. It can be telemetry from dedicated Pragmas, or just all messages received by a current Pragma.

Extentions can be used for integration with the third party systems. Right now out of the box we have HDFS, Rest, and ElasticSearch extension. The REST extension exposes the http and web-socket interfaces, which allows not only to receive data from a cluster, but also to send commands into a cluster as well. In our tests, we were using this feature for drone-controlling from a web page.

Real-World Example

Several drones are randomly flying in a closed area, and a ground control station doesn't allow them to leave this area.

Here we have two pragma types:

- A Drone pragma
- A Manager pragma that works as a ground control station

Drone Pragma Listing

```
object EmbeddedMain extends App with DroneCommands {

  var (vx, vy) = (Random.nextFloat(), Random.nextFloat())

  EmbeddedPragma {
    ctx => {
      case Start =>
        ctx.subscribeHardwareEvents()
        ctx.self ! moveTo(0,0,-10)
        ctx.self ! direction(vx, vy, 0)

      case TelemetryBatch(batch) =>  // TelemetryBatch - message contains drone
        val position = batch.collect { case DronePositionLocal(p) => p }.lastOption
        ctx.listeners foreach ( _ ! position )

      case "turn x" =>
        vx = -vx
        ctx.self ! direction(vx, vy, 0)

      case "turn y" =>
        vy = -vy
        ctx.self ! direction(vx, vy, 0)
    }
  }
}
```

Drone Pragma Listing (cont.)

Drone pragma handles only four messages:

1. **Start** – an internally generated message, sent right after a node is joined to the cluster and is ready to work.
2. **TelemetryBatch** – an internally generated message; this kind of message is sent by HardwareGate and contains hardware telemetry.
3. **“turn x”** – a custom message sent from ground control station; it changes the **X** speed to opposite.
4. **“turn y”** – a custom message sent from ground control station; it changes the **Y** speed to opposite.

Drone Pragma Listing (cont.)

The handler of the “Start” message contains the following code:

ctx.subscribeHardwareEvents() – turns on receiving events from hardware layer

ctx.self ! moveTo(**0**,**0**,-**10**) – sets initial position of drone
ctx.self ! direction(vx, vy, **0**) – sets initial vector of movement

The functions **“moveTo”** and **“direction”** produce special hardware-dedicated messages.

Ground Control Listing

```
object GCMain extends App {

  GRoundControlNode
    .build()
    .addExtention[DroneControlExt]
    .start()
}

class DroneControlExt extends GCExtentions with DroneCommands {

  override def process(manager: ActorRef): Receive = {

    case DronePositionLocal(position) =>
      if (position.x > 100 || position.x < -100) sender() ! "turn x"
      if (position.y > 100 || position.y < -100) sender() ! "turn y"
  }
}
```

Ground Control Listing (cont.)

- The Ground control pragma snippet is even simpler than drone snippet as it handles only one message – “DronePositionLocal”.
- This message is generated by a drone-node and contains drone physical position.
- In the message handler, the drone position is analyzed and appropriate correcting message is sent back to the drone.
- This is all you need, no other configurations required.

Behind the Scene

(How it works)

Pragmukko provides two basic pragma-types:

- **Pragma** – for implementing device-side logic;
- **PragmaSup** (need to be renamed to Manager Node)– for implementing common cluster logic.

In example above, the drone-node is implemented using Pragma.

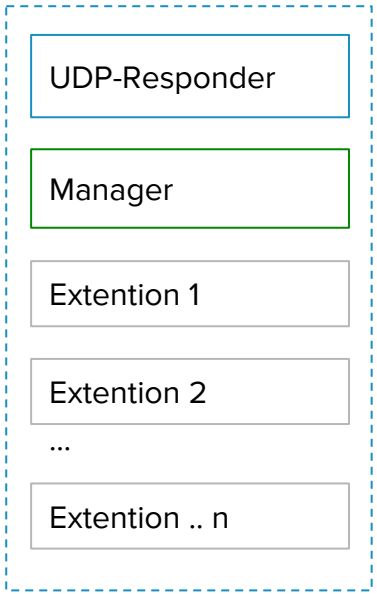
Each of basic node types has a dedicated builder.

PragmaSup

The purpose of PragmaSup node is being the single point of orchestration for other nodes. Also, this node type can be used for obtaining information from other nodes and storing it. For example, storing telemetry from all devices in HDFS for further analysis.

You can run any number of the PragmaSup nodes in a cluster, or even to not run them at all. But it is recommended to have at least 1 node of such type per cluster, as by default, it also runs node-discovering features.

PragmaSup Structure



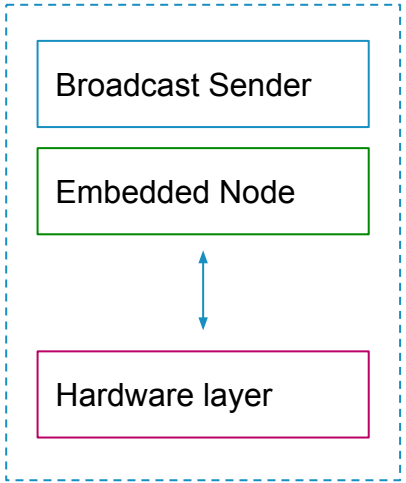
The GroundControl node consists of several Actors within one Akka-node:

UDPResponder is the component of node-discovering subsystem. It's listening to broadcast messages from other nodes and trying to join the cluster.

Manager handles internal Pragmukko messages.

Extentions – a set of user-defined Actors that implement user logic. It's possible to provide any number of Extensions. For example, one extension can write telemetry from all nodes to log file, and the other can implement collision-avoidance algorithm.

Pragma structure



Embedded Node runs on devices and implements device-level logic. This node was developed to be as simple as possible, as it works on small devices.

Broadcast Sender – a component of a node-discovering subsystem.

Embedded Node handles system and user-defined messages.

Hardware layer – a facade for interacting with hardware.

Feature: Service Discovering

Broadcast discovering

New member

We added a service-discovering feature that uses UDP broadcasting for searching new cluster members.

So if nodes share the same cluster-ID, they will automatically join the same cluster.

No seed-node configuration anymore.

Roadmap

- Improving service discovering
  - Zookeeper node discoverer – 1 week
  - Consul node discoverer – 1 week
- Discovering over NAT – 2 weeks
- Integrating with third party solutions
  - Adapter for MongoDB – 1 week
- Implementing analytics framework
  - Adapter for Spark Streaming – 1 week
  - Integration with Apache Flink – 2 weeks
- Improving hardware interaction layer – 2 weeks
- Creating web dashboard – 2 weeks
- Performing deployment and diagnostic tools – 2 weeks