DATABIZ
your solution builder

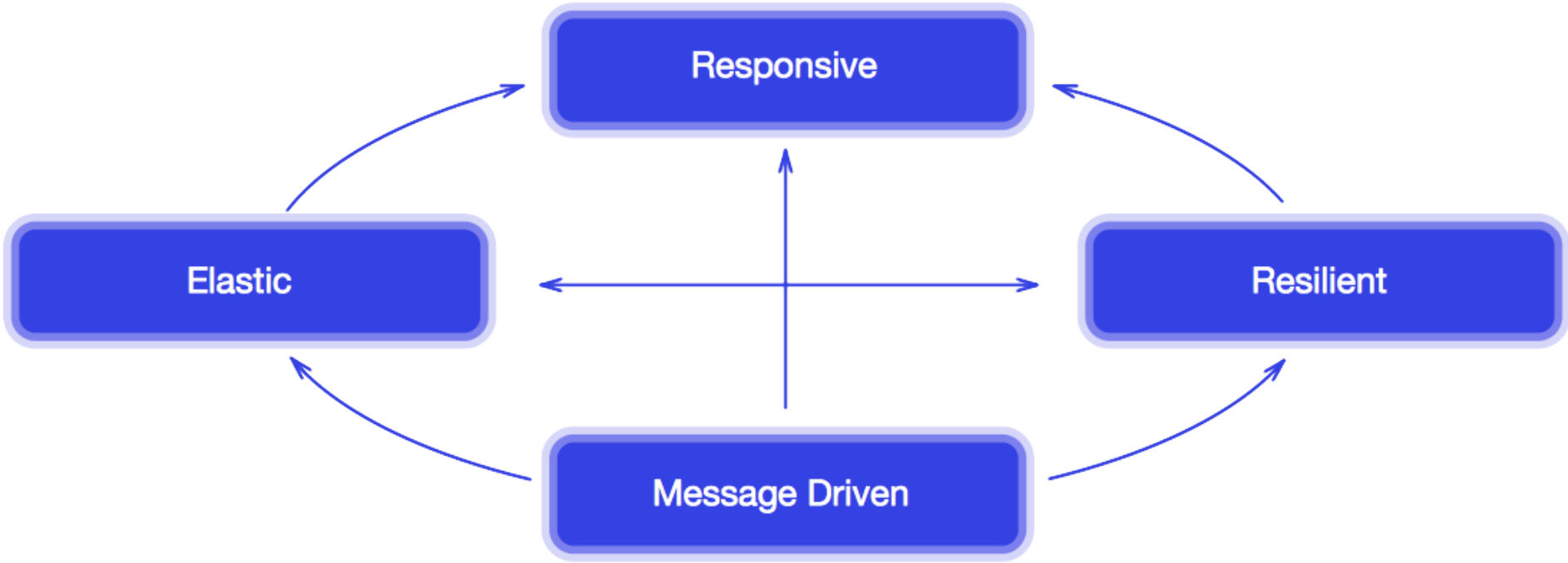
An introduction to reactive programming using Scala and Akka

Roberto Bentivoglio

Milano, 15th January 2015

"Today's demands are simply not met by yesterday's software architectures"

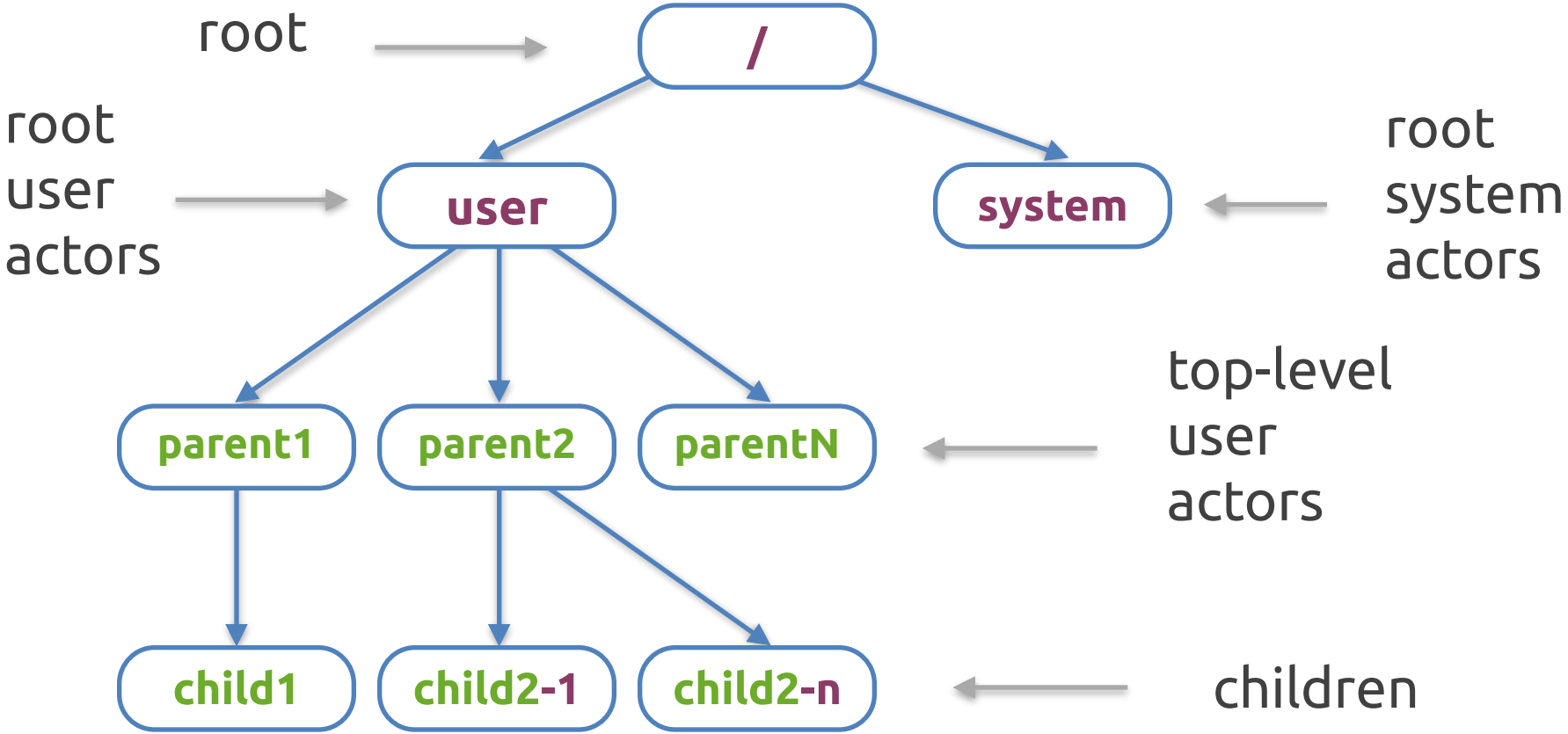
"We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems."



"Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM."

- Event-driven
- Elastic / Scalable
  - Scale out
  - abstraction for transparent distribution
- Resilient
  - Supervisor hierarchies with "let-it-crash" semantics
  - Fault-tolerant and self-healing systems
- Responsive
  - Asynchronous, non-blocking and highly performant event-driven programming mode

"The actor model in computer science is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent computation: in response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received" (Wikipedia)



An ***Actor System*** is a container of actors

Actors are arranged in a hierarchy (a tree)

Actor System

```
// create a new actor system
val system = ActorSystem("reactive-chat-actor-system")
```

```
// shutdown the actor system
system.shutdown()
```

```
// creating a new top-level actor
val user: ActorRef = system.actorOf(Props(new User), "user")
```

- Define the ActorSystem
  - Define it inside `com.databiz.training.actors.Actors.scala` in the `ProductionActorSystem` trait
  - define `chatService`
  - Define the shutdown hook
  - run the tests

An actor is a container for:

- State
- Behavior
- Mailbox
- Children
- Supervisor Strategy.

All of this is encapsulated behind an Actor Reference (ActorRef)


```
class User extends Actor with ActorLogging {

  override def receive: Receive = {
    case message: String =>
      // do something here...
      log.info("Received the message {}", message)
  }
}
```

The receive method defines the behaviour

```
type Receive = PartialFunction[Any, Unit]
```

- Process events and generate responses (or more requests) in an event-driven manner
- Do not block!
- Do not pass mutable objects between actors!
- Do not share internal mutable state!

Akka - fire-and-forget semantics

```
user ! "Hi, wazzup?"
// short form of
user.tell("Hi, wazzup?", context.self)
```

```
final def tell(msg: Any, sender: ActorRef): Unit =
  this.!(msg)(sender)
```

Exercise 2

- Complete first version of `ChatService` actor
  - Implement the protocol for `ChatService`
    - Join
      - `success -> log.info("{} joined the chat", username)`
      - `error -> log.warning(s"The user $username already joined the chat”)`
    - Leave
      - `success -> log.info("{} left the chat", username)`
      - `error -> log.warning(s"The user $username didn't join the chat")`
    - GetUsers

Akka - ask pattern

```
import akka.actor._
import akka.util.Timeout
import scala.concurrent.duration._

implicit val timeout: Timeout = 5 seconds
user ? "Hi, wazzup?"
```

```
def ask(message: Any)
       (implicit timeout: Timeout): Future[Any]
```

An actor path consists of:

- An anchor (to identify the actor system)
- A concatenation of path elements (from root guardian to the designed actor)
- The path elements are the names of the traversed actors and are separated by slashes

```
// local
“akka://reactive-chat-actor-system/user/service-a/worker1"

// remote
"akka.tcp://reactive-chat-actor-system@host.example.com:5678/user/
service-b"
```

Everything in Akka is designed to work in a distributed setting: all interactions of actors use purely message passing and everything is asynchronous

Akka Remoting is designed for communication in a peer-to-peer fashion and it has limitations for client-server setups

```
context.actorSelection("akka.tcp://rchat-actor-system@10.10.10.2:2552/user/actor1")
```

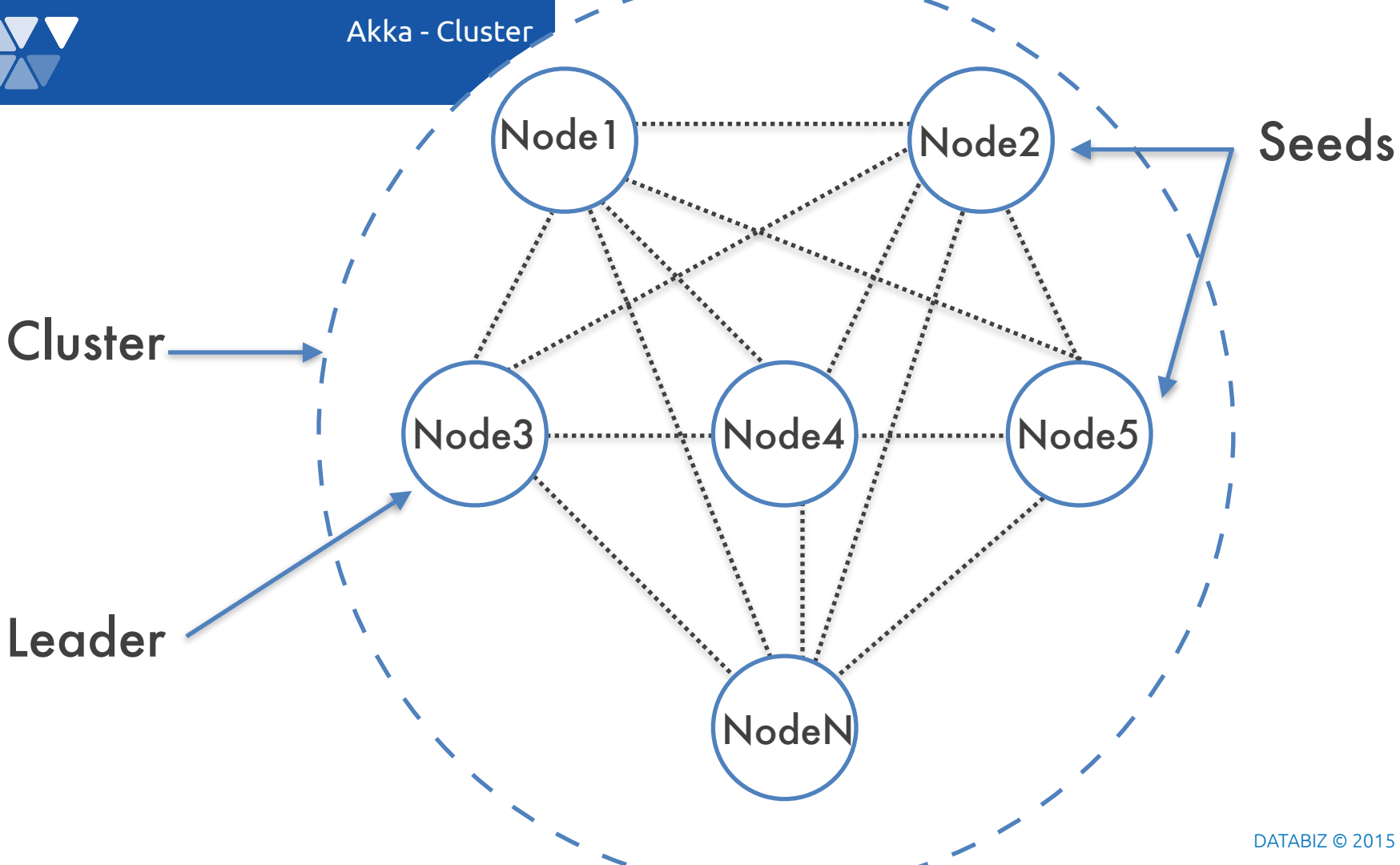
Exercise 3

- Complete new version to the service
  - Add to ChatService the protocol
    - `DeliverMessage(from: String, to: String, message: String)`
  - `Refactor the user management: how could we memorize users ActorRef?`
  - Implement the `User actor, com.databiz.training.core.User.scala`
    - `use the following protocol:`
      - `case class Message(from: String, text: String) extends UserProtocol`

Akka Cluster provides a:

- Fault-tolerant, decentralised, peer-to-peer based cluster membership service
- With no single point of failure or single point of bottleneck
- It uses gossip protocols and an automatic failure detector

- **Node:** A logical member of a cluster. There could be multiple nodes on a physical machine. Defined by a hostname:port:uid tuple.
- **Cluster:** A set of nodes joined together through the membership service.
- **Leader:** A single node in the cluster that acts as the leader. It is simply the first node in sorted order that is able to take the leadership role
- **Seed nodes:** configured contact points for new nodes joining the cluster



```
akka {
  actor.provider = "akka.cluster.ClusterActorRefProvider"
  extensions = ["akka.contrib.pattern.DistributedPubSubExtension"]
  remote.netty.tcp {
    hostname = "localhost"
    port = 2552
  }
  cluster {
    seed-nodes=["akka.tcp://reactive-chat-server@localhost:
2552","akka.tcp://application@localhost:2553"]
    roles = [backend]
  }
}
```

```
// create a new cluster
val cluster = Cluster(system)
```

- Cluster singleton: one actor of a certain type running somewhere in the cluster
- Cluster sharding: distribute actors across several nodes in the cluster and want to be able to interact with them using their logical identifier, but without having to care about their physical location in the cluster, which might also change over time

- Distributed Publish Subscribe (topic): This pattern provides a mediator actor, `akka.contrib.pattern.DistributedPubSubMediator`, that manages a registry of actor references and replicates the entries to peer actors among all cluster nodes or a group of nodes tagged with a specific role.
- Cluster Client: An actor system that is not part of the cluster can communicate with actors somewhere in the cluster via this ClusterClient

Biblio & Links

- Reactive Manifesto - http://www.reactivemanifesto.org/
- Scala & Akka - http://www.typesafe.com/
- Scala - Programming in Scala 2nd edition - http://www.artima.com/shop/programming_in_scala_2ed
- Akka documentation - http://akka.io/docs/
- Akka team blog - http://letitcrash.com/
- Akka concurrency - http://www.artima.com/shop/akka_concurrency
- Akka in Action - http://www.manning.com/roestenburg/
- Activator example on mediator pattern - http://typesafe.com/activator/template/reactive-maps

DATABIZ

your solution builder

We're hiring!

info@databiz.it

Thanks!

Q&A

roberto.bentivoglio@databiz.it
Follow me on Twitter! @robbenti

DATABIZ
your solution builder