CAKESOLUTIONS
ENTERPRISE SOFTWARE SERVICES

MANCHESTER LONDON NEW YORK

Chaos Engineering and Runtime Monitoring of Distributed Reactive Systems

Dr. Carl Pulley
Cake Solutions @cakesolutions

Chaos Engineering

Chaos Engineering

- Emphasises an empirical approach to testing and monitoring distributed systems

Chaos Engineering

- Emphasises an empirical approach to testing and monitoring distributed systems
- Typical Chaos Experiment
  - define views of a system (e.g. by monitoring a set of system metrics)
  - randomly inject faults (e.g. by killing containers or changing networking behaviour)

Chaos Engineering

- Emphasises an empirical approach to testing and monitoring distributed systems
- Typical Chaos Experiment
  - define views of a system (e.g. by monitoring a set of system metrics)
  - randomly inject faults (e.g. by killing containers or changing networking behaviour)
- Infer weaknesses using deviations from expected or steady-state behaviour

Chaos Engineering

- Emphasises an empirical approach to testing and monitoring distributed systems
- Typical Chaos Experiment
  - define views of a system (e.g. by monitoring a set of system metrics)
  - randomly inject faults (e.g. by killing containers or changing networking behaviour)
- Infer weaknesses using deviations from expected or steady-state behaviour
- Potentially apply to production systems!

The Goal

The Goal

- Verify the correctness of a distributed system

The Goal

- Verify the correctness of a distributed system
- Typical verification approaches
  - Testing
  - Static analysis
  - Formal/model based verification

What Happens Next?

What Happens Next?

- Excellent, we now feel confident that the application is correct

What Happens Next?

- Excellent, we now feel confident that the application is correct
- But what happens next?

What About Correctness?

CAKESOLUTIONS

What About Correctness?

- You Docker containerise your code

What About Correctness?

- You Docker containerise your code
- DevOps wrap your code up with provisioning code, inject processes into containers, add monitoring and instrumentation code, etc.

What About Correctness?

- You Docker containerise your code
- DevOps wrap your code up with provisioning code, inject processes into containers, add monitoring and instrumentation code, etc.
- DevOps then deploy to and orchestrate commodity cloud based hardware

What About Correctness?

- You Docker containerise your code
- DevOps wrap your code up with provisioning code, inject processes into containers, add monitoring and instrumentation code, etc.
- DevOps then deploy to and orchestrate commodity cloud based hardware
- The data centre dynamically maintains/relocates your code

What About Correctness?

- You Docker containerise your code
- DevOps wrap your code up with provisioning code, inject processes into containers, add monitoring and instrumentation code, etc.
- DevOps then deploy to and orchestrate commodity cloud based hardware
- The data centre dynamically maintains/relocates your code
- Human intervention may occur during crisis scenarios

What About Correctness?

- You Docker containerise your code
- DevOps wrap your code up with provisioning code, inject processes into containers, add monitoring and instrumentation code, etc.
- DevOps then deploy to and orchestrate commodity cloud based hardware
- The data centre dynamically maintains/relocates your code
- Human intervention may occur during crisis scenarios
- Do you still feel confident that the application is correct?

The Problem

The Problem

- Deploy our proven correct application code into dynamic environments

The Problem

- Deploy our proven correct application code into dynamic environments
  - we have little or no real control over these environments!

The Problem

- Deploy our proven correct application code into dynamic environments
  - we have little or no real control over these environments!
  - deployment typically uses unverified processes

The Problem

- Deploy our proven correct application code into dynamic environments
  - we have little or no real control over these environments!
  - deployment typically uses unverified processes
- Potential for unanticipated failure is now highly likely

The Problem

- Deploy our proven correct application code into dynamic environments
  - we have little or no real control over these environments!
  - deployment typically uses unverified processes
- Potential for unanticipated failure is now highly likely
  - embrace and test for failure!

docker-compose-testkit

docker-compose-testkit

- Open-source Scala library (under development)
  - (optionally) deploys, orchestrates and instruments Docker container code
  - agnostic of deployment environment
  - composable behaviour properties
  - reusable Chaos experiments *easily* defined

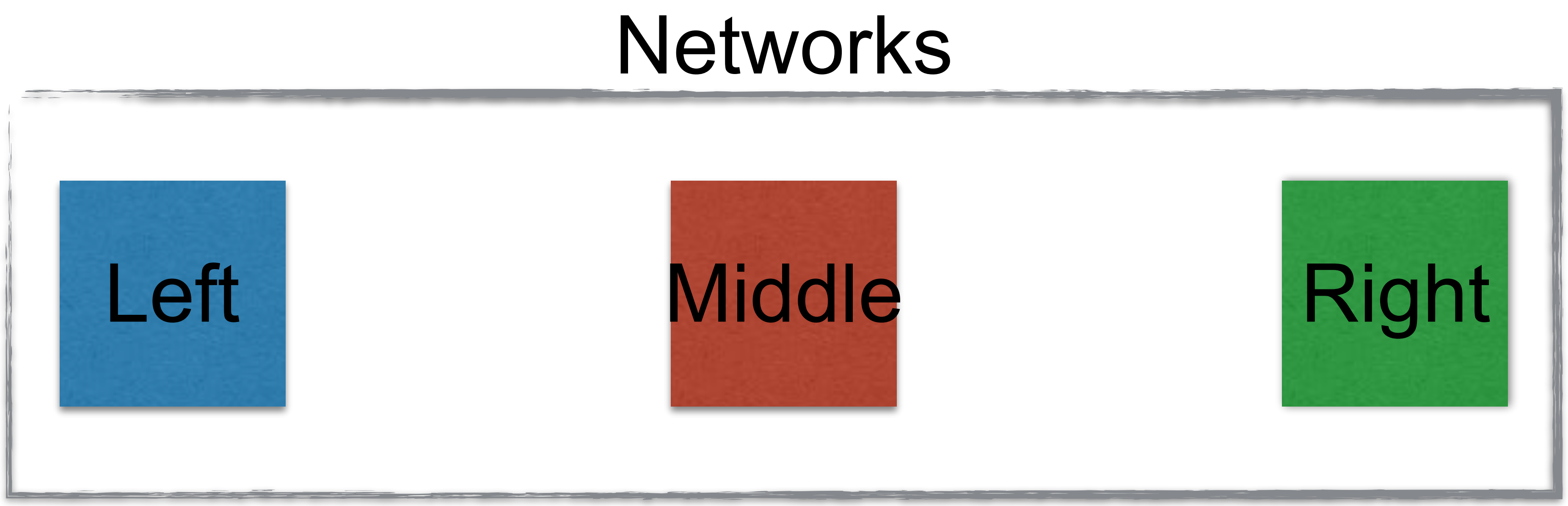
templates

templates

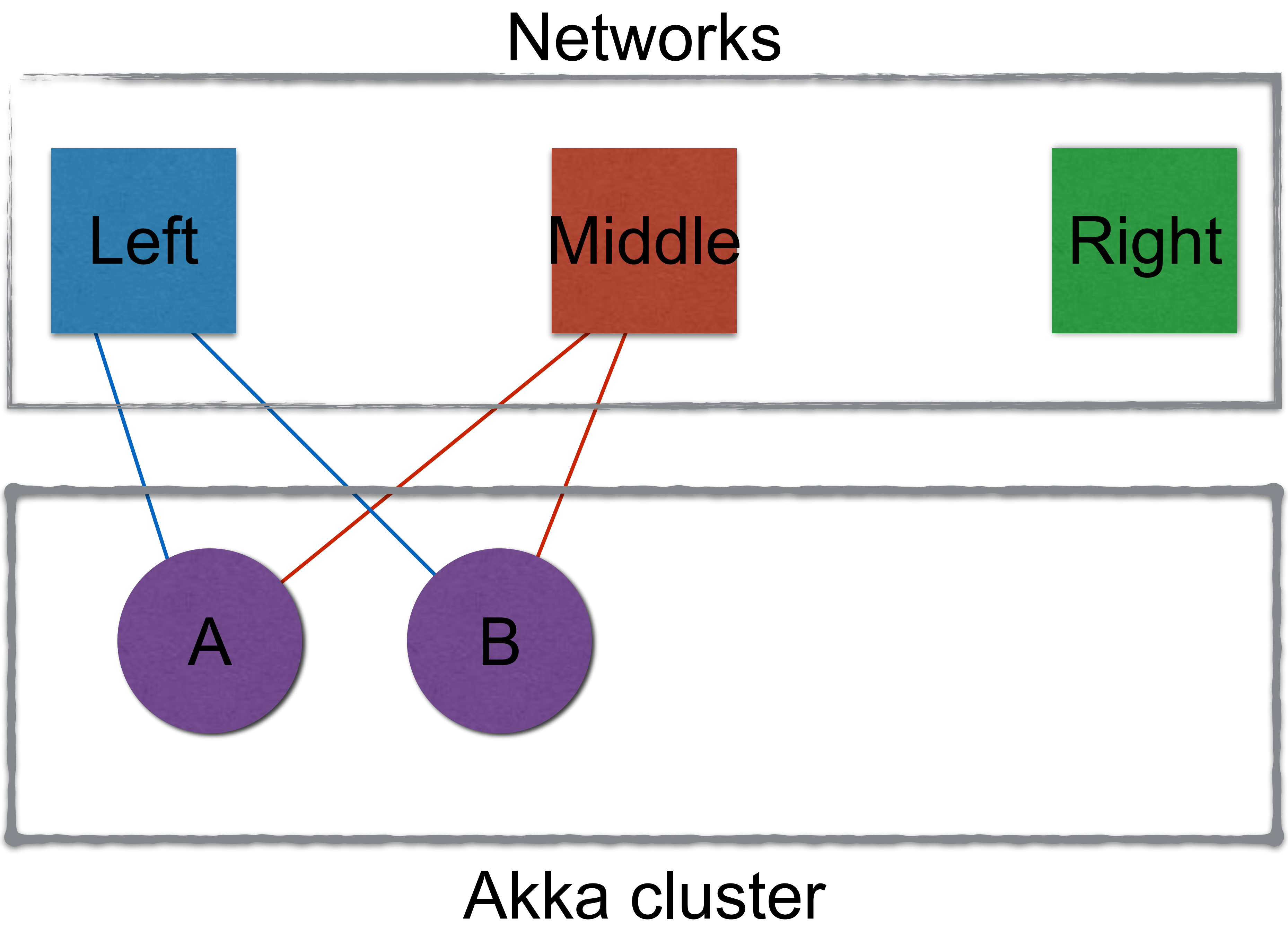
- Specify Docker image instrumentation layers
  - injects code and resources into image
- Specify library code for using instrumentation
  - code needs to use side effects
  - maintain a functional core using extensible effects (i.e. Eff)
  - sensor events represented using Monix Observables

Demo Overview

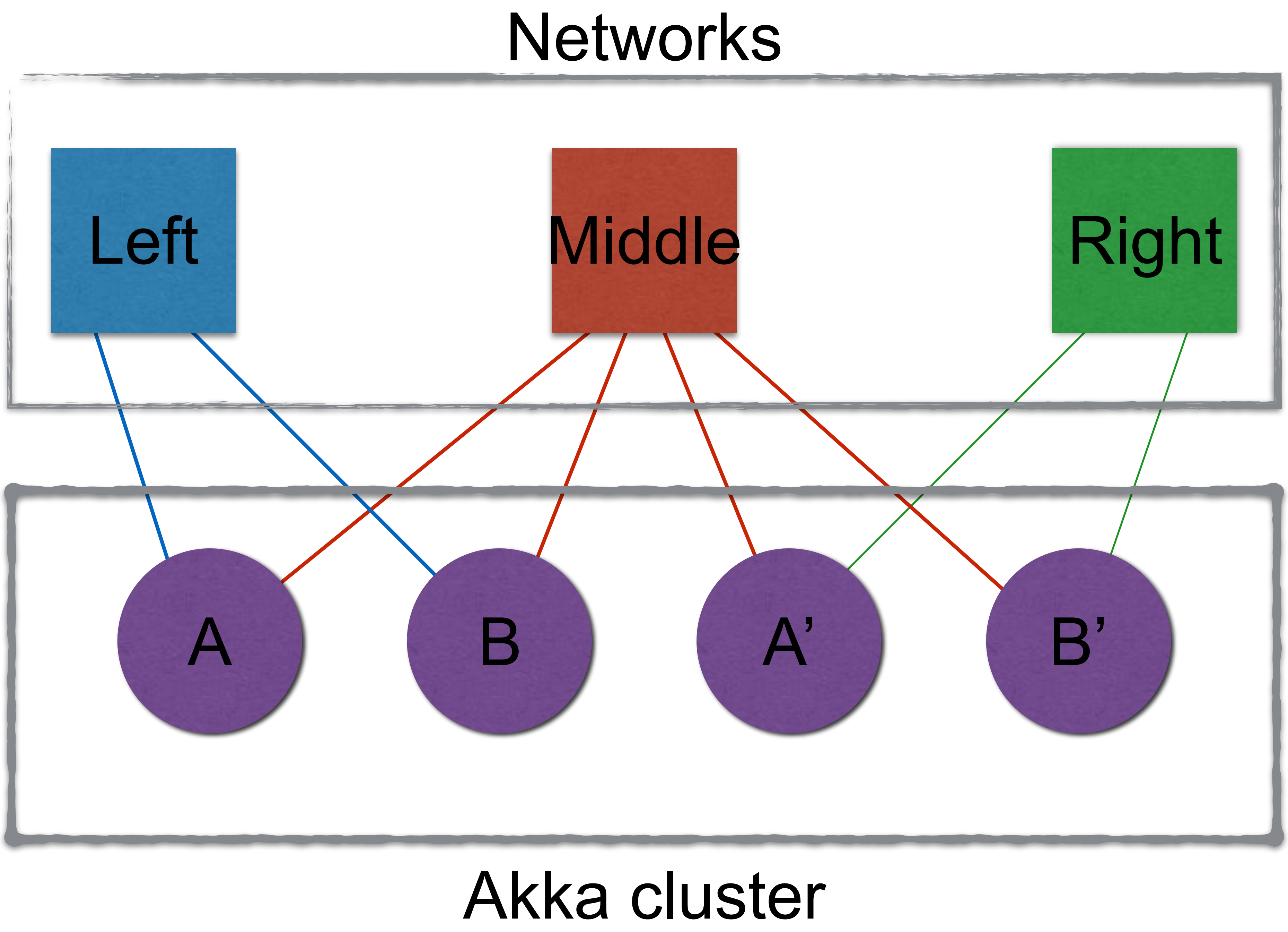
Demo Overview



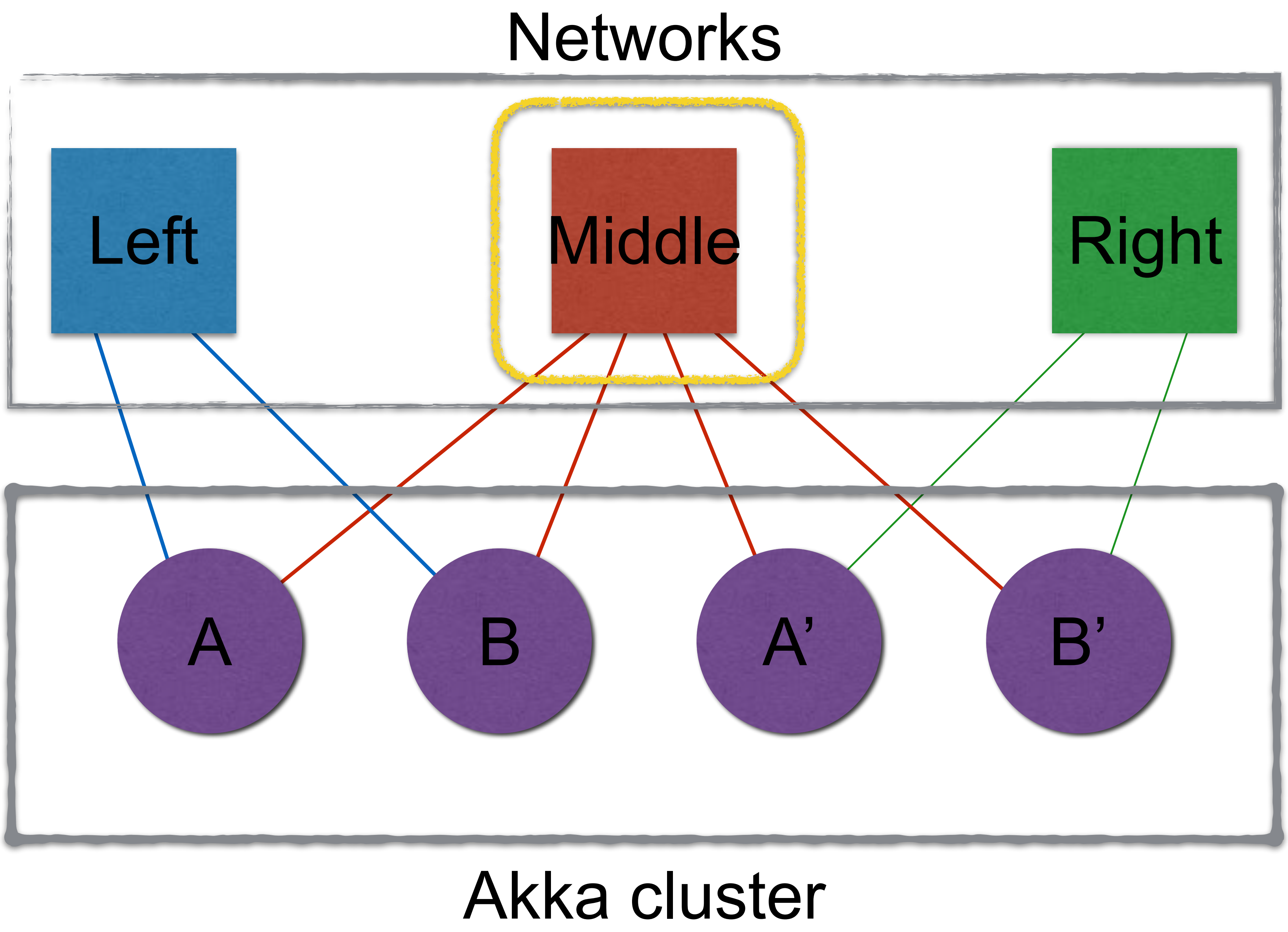
Demo Overview



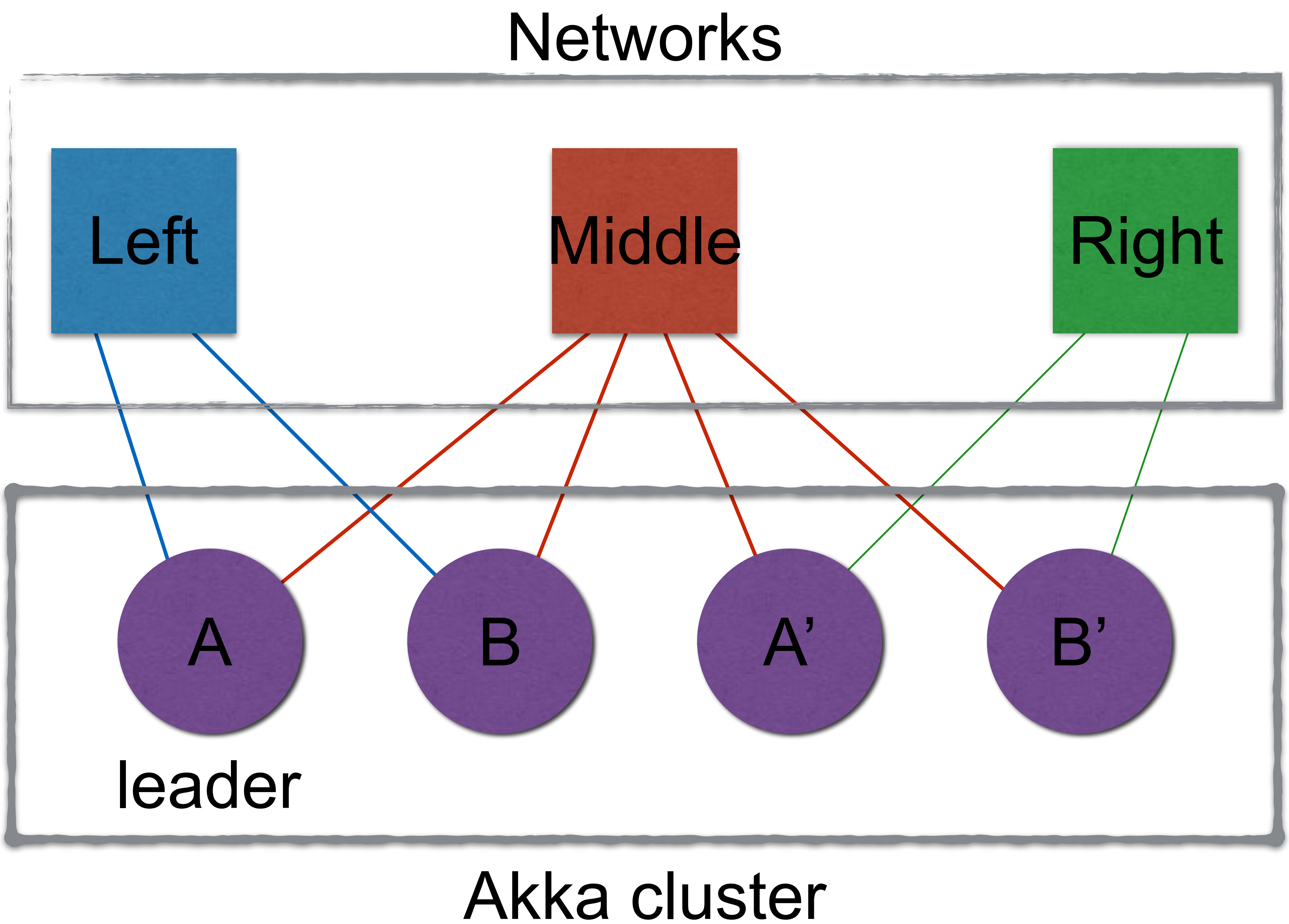
Demo Overview



Demo Overview



Demo Overview



Auto-downing Enabled!!

Demo Overview

```scala
def clusterNode(name: String, network1: String, network2: String): String =
  s"""$name:
      |    template:
      |      resources:
      |        - cakesolutions.docker.jmx.akka
      |        - cakesolutions.docker.network.default.linux
      |      image: docker-compose-testkit-tests:$version
      |    environment:
      |      AKKA_HOST: $name
      |      AKKA_PORT: 2552
      |      CLUSTER_SEED_NODE: "akka.tcp://TestCluster@A:2552"
      |    expose:
      |      - 2552
      |    networks:
      |      - $network1
      |      - $network2
   """.stripMargin

val yaml = DockerComposeString(
  s"""version: '2'
       |
       |services:
       |  ${clusterNode("A", "left", "middle")}
       |  ${clusterNode("B", "left", "middle")}
       |  ${clusterNode("A'", "right", "middle")}
       |  ${clusterNode("B'", "right", "middle")}
       |
       |networks:
       |  left:
       |  middle:
       |  right:
  """.stripMargin
)
```

Demo Overview

```scala
def clusterNode(name: String, network1: String, network2: String): String =
  s"""$name:
     |    template:
     |      resources:
     |        - cakesolutions.docker.jmx.akka
     |        - cakesolutions.docker.network.default.linux
     |      image: docker-compose-testkit-tests:$version
     |    environment:
     |      AKKA_HOST: $name
     |      AKKA_PORT: 2552
     |      CLUSTER_SEED_NODE: "akka.tcp://TestCluster@A:2552"
     |    expose:
     |      - 2552
     |    networks:
     |      - $network1
     |      - $network2
   """.stripMargin

val yaml = DockerComposeString(
  s"""version: '2'
      |
      |services:
      |  ${clusterNode("A", "left", "middle")}
      |  ${clusterNode("B", "left", "middle")}
      |  ${clusterNode("A'", "right", "middle")}
      |  ${clusterNode("B'", "right", "middle")}
      |
      |networks:
      |  left:
      |  middle:
      |  right:
  """.stripMargin
)
```

Demo Overview

```scala
def clusterNode(name: String, network1: String, network2: String): String =
  s"""$name:
      |    template:
      |      resources:
      |        - cakesolutions.docker.jmx.akka
      |        - cakesolutions.docker.network.default.linux
      |      image: docker-compose-testkit-tests:$version
      |    environment:
      |      AKKA_HOST: $name
      |      AKKA_PORT: 2552
      |      CLUSTER_SEED_NODE: "akka.tcp://TestCluster@A:2552"
      |    expose:
      |      - 2552
      |    networks:
      |      - $network1
      |      - $network2
   """.stripMargin

val yaml = DockerComposeString(
  s"""version: '2'
       |
       |services:
       |  ${clusterNode("A", "left", "middle")}
       |  ${clusterNode("B", "left", "middle")}
       |  ${clusterNode("A'", "right", "middle")}
       |  ${clusterNode("B'", "right", "middle")}
       |
       |networks:
       |  left:
       |  middle:
       |  right:
  """.stripMargin
)
```

Demo Overview

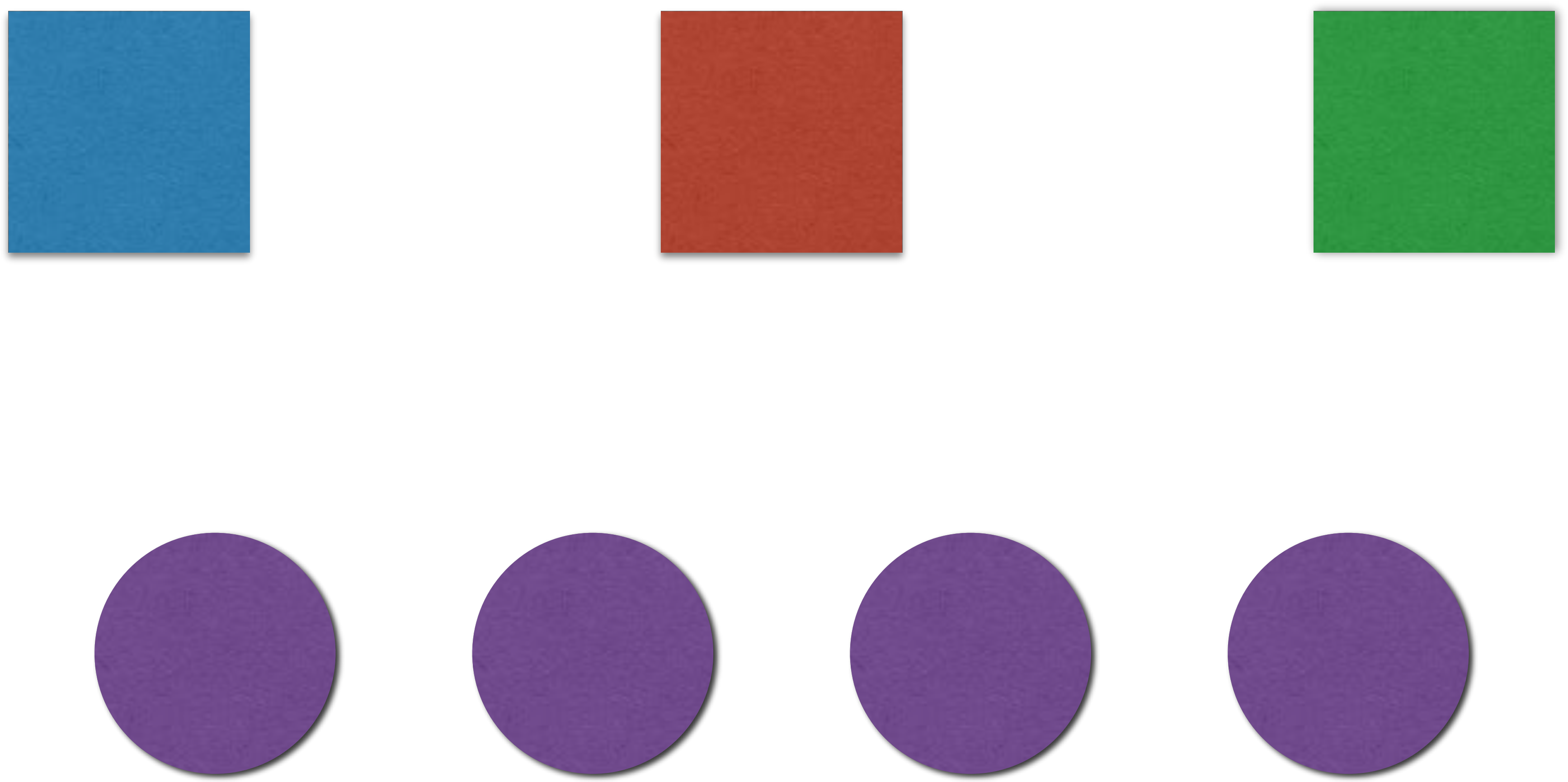
```
def clusterNode(name: String, network1: String, network2: String): String =
  s"""$name:
     |    template:
     |      resources:
     |        - cakesolutions.docker.jmx.akka
     |        - cakesolutions.docker.network.default.linux
     |      image: docker-compose-testkit-tests:$version
     |    environment:
     |      AKKA_HOST: $name
     |      AKKA_PORT: 2552
     |      CLUSTER_SEED_NODE: "akka.tcp://TestCluster@A:2552"
     |    expose:
     |      - 2552
     |    networks:
     |      - $network1
     |      - $network2
   """.stripMargin

val yaml = DockerComposeString(
  s"""version: '2'
      |
      |services:
      |  ${clusterNode("A", "left", "middle")}
      |  ${clusterNode("B", "left", "middle")}
      |  ${clusterNode("A'", "right", "middle")}
      |  ${clusterNode("B'", "right", "middle")}
      |
      |networks:
      |  left:
      |  middle:
      |  right:
  """.stripMargin
)
```

Demo Overview

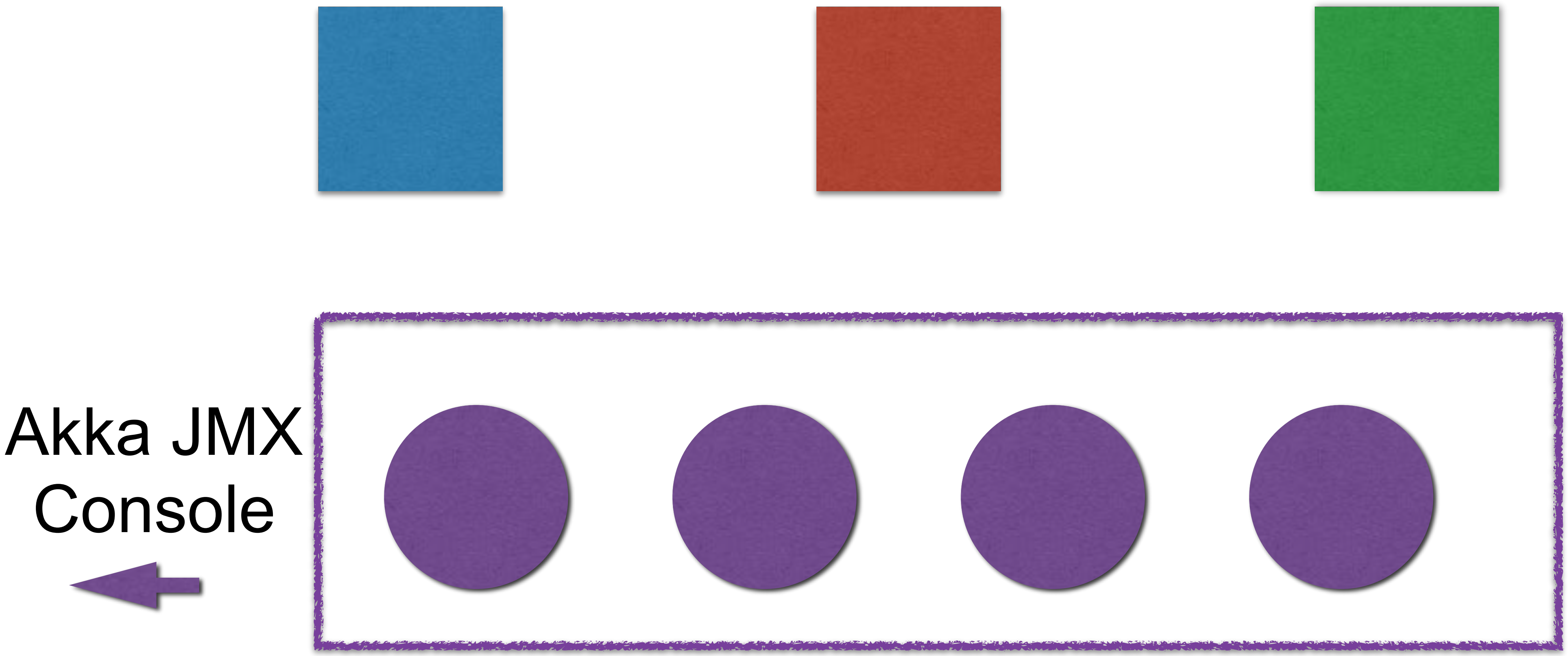
```
def clusterNode(name: String, network1: String, network2: String): String =
  s"""$name:
      |    template:
      |      resources:
      |        - cakesolutions.docker.jmx.akka
      |        - cakesolutions.docker.network.default.linux
      |      image: docker-compose-testkit-tests:$version
      |    environment:
      |      AKKA_HOST: $name
      |      AKKA_PORT: 2552
      |      CLUSTER_SEED_NODE: "akka.tcp://TestCluster@A:2552"
      |    expose:
      |      - 2552
      |    networks:
      |      - $network1
      |      - $network2
   """.stripMargin

val yaml = DockerComposeString(
  s"""version: '2'
       |
       |services:
       |  ${clusterNode("A", "left", "middle")}
       |  ${clusterNode("B", "left", "middle")}
       |  ${clusterNode("A'", "right", "middle")}
       |  ${clusterNode("B'", "right", "middle")}
       |
       |networks:
       |  left:
       |  middle:
       |  right:
  """.stripMargin
)
```

immutable container

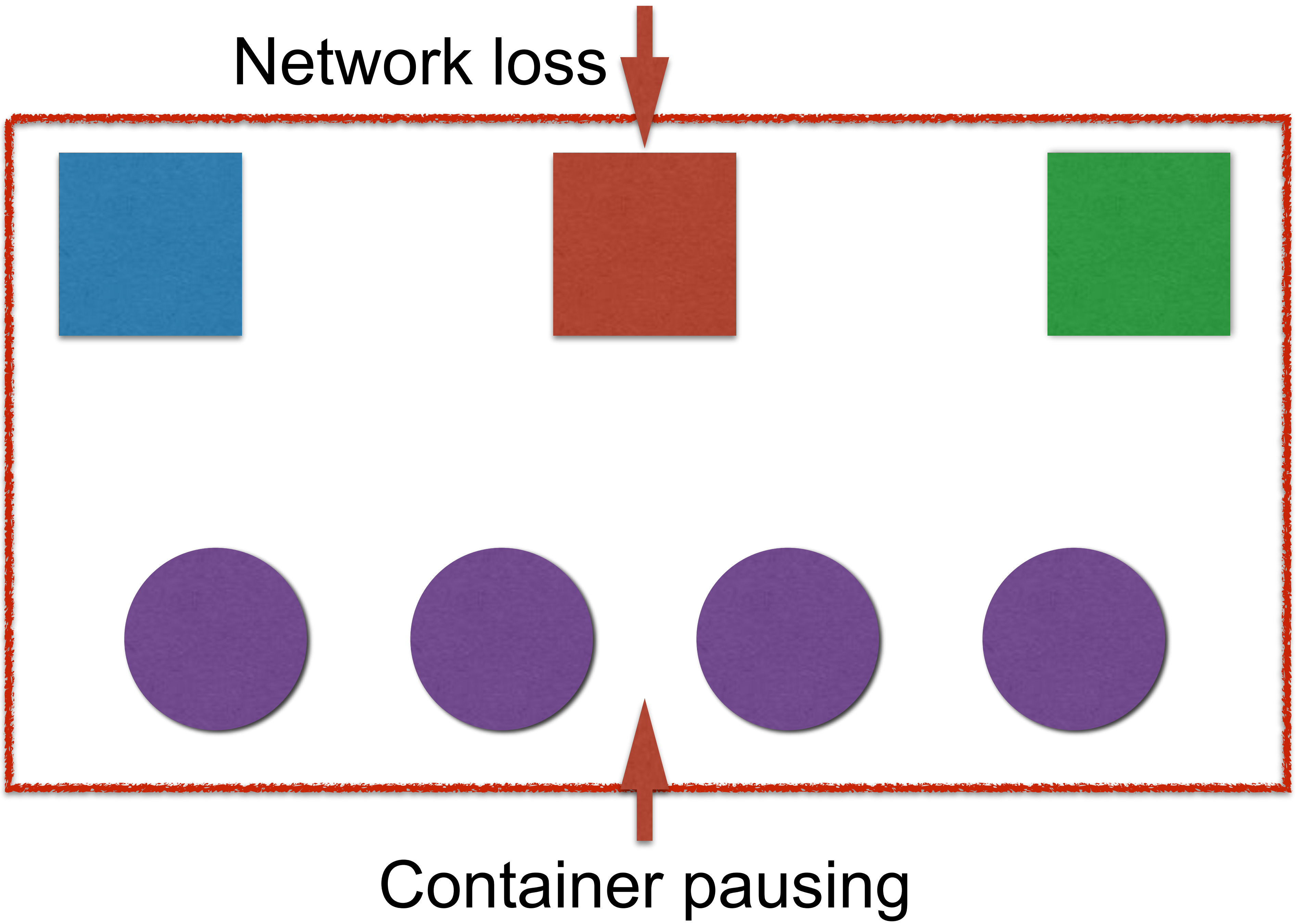
Demo Overview



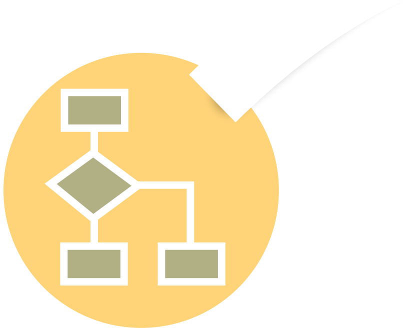
Demo Overview



Demo Overview

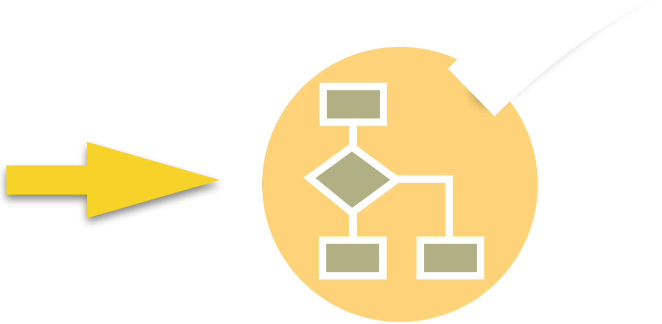


Chaos Experiment

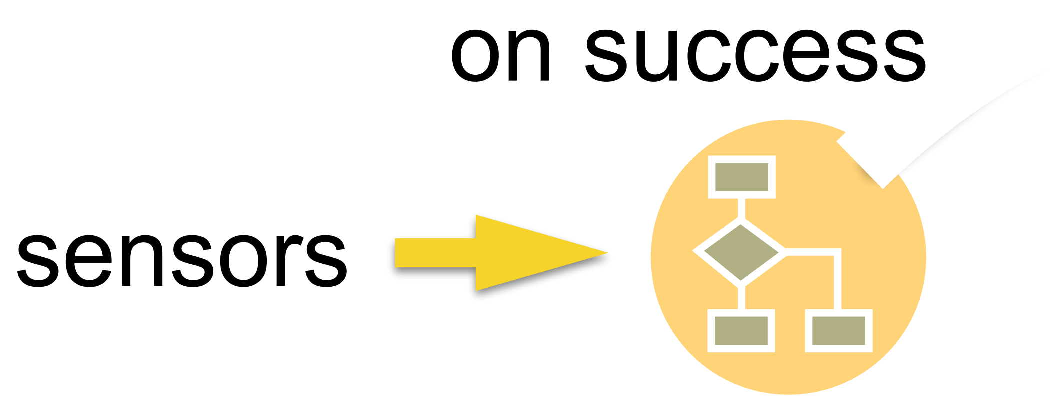


Chaos Experiment

sensors

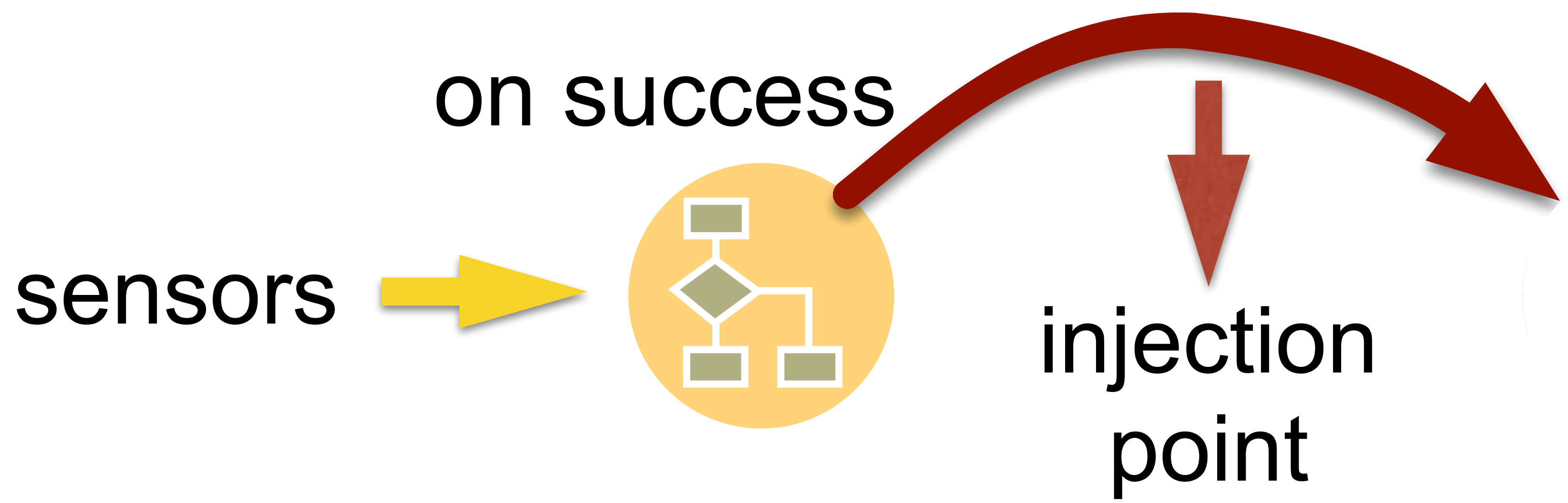


Chaos Experiment



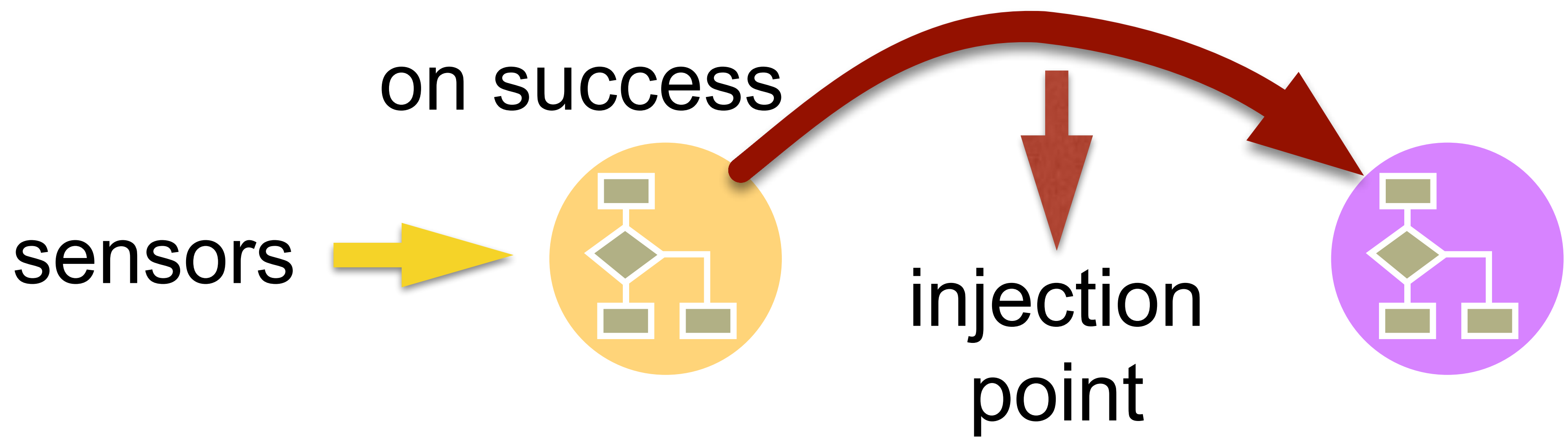
- Interpret this as follows:
  - when we successfully detect behaviour

Chaos Experiment



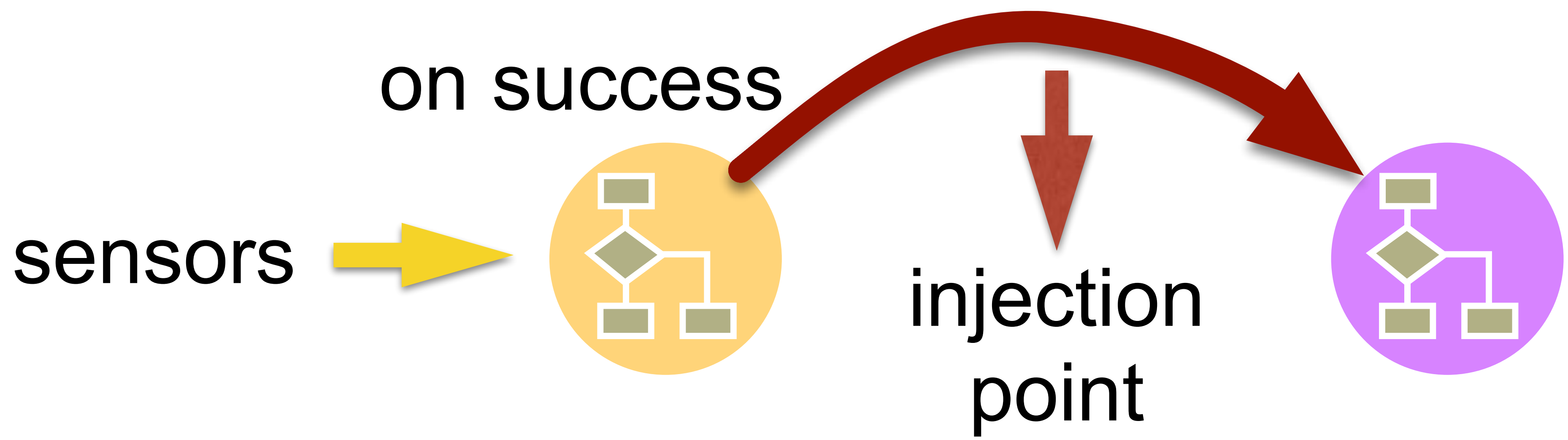
- Interpret this as follows:
  - when we successfully detect behaviour
  - action may happen

Chaos Experiment



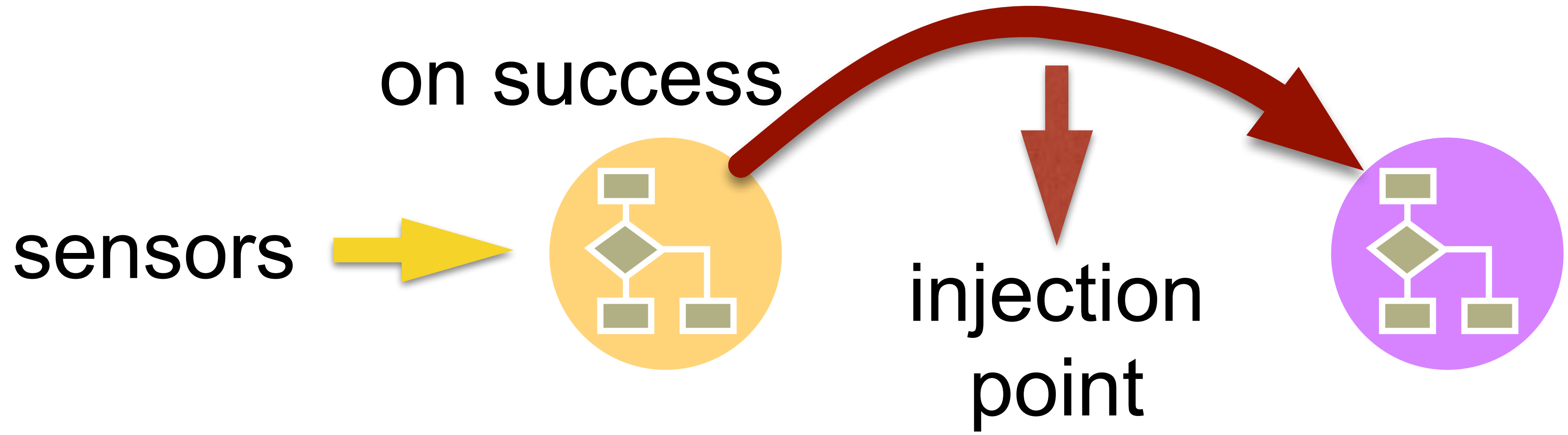
- Interpret this as follows:
  - when we successfully detect behaviour
  - action may happen
  - and we start monitoring for behaviour

Chaos Experiment



- Interpret this as follows:
  - when we successfully detect behaviour
  - action may happen
  - and we start monitoring for behaviour
- Chaining such relations allows the fault injection search space to be described

Chaos Experiment



- Interpret this as follows:
  - when we successfully detect behaviour
  - action may happen
  - and we start monitoring for behaviour
- Chaining such relations allows the fault injection search space to be described

Demo Overview

```
def experiment[M: _jvm: _network: ..](implicit ..): Eff[M, Notify] =
  for {









  } yield Accept()
```

Demo Overview

```
def experiment[M: _jvm: _network: ..](implicit ..): Eff[M, Notify] =
  for {
    obs1 <- ???
    ..







  } yield Accept()
```

Demo Overview

```
def experiment[M: _jvm: _network: ..](implicit ..): Eff[M, Notify] =
  for {
    obs1 <- ???
    ..
    _ <- jvmGC(JvmGCStart)(rightA)
    _ = note("A' JVM GC pause starts")
    obs2 <- ???
    ..







  } yield Accept()
```

Demo Overview

```scala
def experiment[M: _jvm: _network: ..](implicit ..): Eff[M, Notify] =
  for {
    obs1 <- ???
    ..
    _ <- jvmGC(JvmGCStart)(rightA)
    _ = note("A' JVM GC pause starts")
    obs2 <- ???
    ..
    _ <- jvmGC(JvmGCEnd)(rightA)
    _ = note("A' JVM GC pause ends")
    obs3 <- ???
    ..




  } yield Accept()
```

Demo Overview

```scala
def experiment[M: _jvm: _network: ..](implicit ..): Eff[M, Notify] =
  for {
    obs1 <- ???
    ..
    _ <- jvmGC(JvmGCStart)(rightA)
    _ = note("A' JVM GC pause starts")
    obs2 <- ???
    ..
    _ <- jvmGC(JvmGCEnd)(rightA)
    _ = note("A' JVM GC pause ends")
    obs3 <- ???
    ..
    _ <- impair(Loss("100%"))("middle")
    _ = note("partition into left and right networks")
    obs4 <- ???
    ..
  } yield Accept()
```

Demo Overview

```
def inCluster(nodes: Address*): Monitor[Boolean, AkkaClusterState] = {
  Monitor(false) {
    case false => {
      case Observe(AkkaClusterState(_, members, unreachable))
        if unreachable.isEmpty && nodes.forall(node => members.filter(_.status ==
Up).exists(_.address == node)) =>
        Goto(true, 3.seconds)
    }

    case true => {
      case Observe(AkkaClusterState(_, _, unreachable)) if unreachable.nonEmpty =>
        Stop(Fail())
      case Observe(AkkaClusterState(_, members, _))
        if nodes.exists(node => members.filter(_.status == Up).forall(_.address !=
node)) =>
        Stop(Fail())
      case StateTimeout =>
        Stop(Accept())
    }
  }
}
```

Demo Overview

Monitor State

```
def inCluster(nodes: Address*): Monitor[Boolean, AkkaClusterState] = {
  Monitor(false) { Initial State
    case false => {
      case Observe(AkkaClusterState(_, members, unreachable))
        if unreachable.isEmpty && nodes.forall(node => members.filter(_.status ==
Up).exists(_.address == node)) =>
        Goto(true, 3.seconds)
    }

    case true => {
      case Observe(AkkaClusterState(_, _, unreachable)) if unreachable.nonEmpty =>
        Stop(Fail())
      case Observe(AkkaClusterState(_, members, _))
        if nodes.exists(node => members.filter(_.status == Up).forall(_.address !=
node)) =>
        Stop(Fail())
      case StateTimeout =>
        Stop(Accept())
    }
  }
}
```

Demo Overview

Observed Event

```
def inCluster(nodes: Address*): Monitor[Boolean, AkkaClusterState] = {
  Monitor(false) {
    case false => {
      case Observe(AkkaClusterState(_, members, unreachable))
        if unreachable.isEmpty && nodes.forall(node => members.filter(_.status ==
Up).exists(_.address == node)) =>
        Goto(true, 3.seconds)
    }

    case true => {
      case Observe(AkkaClusterState(_, _, unreachable)) if unreachable.nonEmpty =>
        Stop(Fail())
      case Observe(AkkaClusterState(_, members, _))
        if nodes.exists(node => members.filter(_.status == Up).forall(_.address !=
node)) =>
        Stop(Fail())
      case StateTimeout =>
        Stop(Accept())
    }
  }
}
```

Demo Overview

```
def inCluster(nodes: Address*): Monitor[Boolean, AkkaClusterState] = {
  Monitor(false) {
    case false => {
      case Observe(AkkaClusterState(_, members, unreachable))
        if unreachable.isEmpty && nodes.forall(node => members.filter(_.status ==
Up).exists(_.address == node)) =>
        Goto(true, 3.seconds)
    }

    case true => {
      case Observe(AkkaClusterState(_, _, unreachable)) if unreachable.nonEmpty =>
        Stop(Fail())
      case Observe(AkkaClusterState(_, members, _))
        if nodes.exists(node => members.filter(_.status == Up).forall(_.address !=
node)) =>
        Stop(Fail())
      case StateTimeout =>
        Stop(Accept())
    }
  }
}
```

Timeout Event

CAKESOLUTIONS

Demo Overview

```
def experiment[M: _jvm: _network: ..](implicit ..): Eff[M, Notify] =
  for {












  } yield Accept()
```

Demo Overview

```
def experiment[M: _jvm: _network: ..](implicit ..): Eff[M, Notify] =
  for {
    obs1 <- jmx(inCluster(leftA, leftB, rightA, rightB))(leftA)
    _ <- check(isAccepting(obs1))
    _ = note("cluster formed, is stable and A an available leader")
    ..
  } yield Accept()
```

Demo Overview

```
def experiment[M: _jvm: _network: ..](implicit ..): Eff[M, Notify] =
  for {
    obs1 <- jmx(inCluster(leftA, leftB, rightA, rightB))(leftA)
    _ <- check(isAccepting(obs1))
    _ = note("cluster formed, is stable and A an available leader")
    ..
    obs2 <- jmx(unreachable(rightA))(leftA)
    _ <- check(isAccepting(obs2))
    _ = note("A' is unreachable")
    ..















  } yield Accept()
```

Demo Overview

```
def experiment[M: _jvm: _network: ..](implicit ..): Eff[M, Notify] =
  for {
    obs1 <- jmx(inCluster(leftA, leftB, rightA, rightB))(leftA)
    _ <- check(isAccepting(obs1))
    _ = note("cluster formed, is stable and A an available leader")
    ..
    obs2 <- jmx(unreachable(rightA))(leftA)
    _ <- check(isAccepting(obs2))
    _ = note("A' is unreachable")
    ..
    obs3 <- jmx(inCluster(leftA, leftB, rightA, rightB))(leftA)
    _ <- check(isAccepting(obs3))
    _ = note("cluster stabilised with A' as a member")
    ..




  } yield Accept()
```

Demo Overview

```
def experiment[M: _jvm: _network: ..](implicit ..): Eff[M, Notify] =
  for {
    obs1 <- jmx(inCluster(leftA, leftB, rightA, rightB))(leftA)
    _ <- check(isAccepting(obs1))
    _ = note("cluster formed, is stable and A an available leader")
    ..
    obs2 <- jmx(unreachable(rightA))(leftA)
    _ <- check(isAccepting(obs2))
    _ = note("A' is unreachable")
    ..
    obs3 <- jmx(inCluster(leftA, leftB, rightA, rightB))(leftA)
    _ <- check(isAccepting(obs3))
    _ = note("cluster stabilised with A' as a member")
    ..
    obs4 <- jmx(inCluster(leftA, leftB))(leftA)
              && jmx(inCluster(leftA, leftB))(leftB)
              && jmx(inCluster(rightA))(rightA)
              && jmx(inCluster(rightB))(rightB)
    _ <- check(isAccepting(obs4))
    _ = note("cluster split brains into 3 clusters: A & B; A'; B'")
  } yield Accept()
```

Demo Overview

Demo Overview

```
type Model = Fx.fx5[JmxAction, NetworkAction, ..]
```

Demo Overview

```scala
val cluster = Map(
  leftA -> compose.service(leftA).docker.head,
  leftB -> compose.service(leftB).docker.head,
  rightA -> compose.service(rightA).docker.head,
  rightB -> compose.service(rightB).docker.head
)
```

Demo Overview

```
experiment[Model].runJvm(cluster).runNetwork…
```

Demo Overview

```
experiment[Model].runJvm(cluster).runNetwork…
```

Demo Time!

Conclusion

Conclusion

- Seen how to model a variety of distributed systems extensionally

Conclusion

- Seen how to model a variety of distributed systems extensionally
- Taken steady state behavioural specifications and developed a composable DSL for experiments
  - by relating data injection traces to observed system behaviour

Conclusion

- Seen how to model a variety of distributed systems extensionally
- Taken steady state behavioural specifications and developed a composable DSL for experiments
  - by relating data injection traces to observed system behaviour
- Applied techniques to a range of fault-injection scenarios

Next Steps

Next Steps

- Prove out library on a range of real-world examples
  - avoid closed work assumptions

Next Steps

- Prove out library on a range of real-world examples
  - avoid closed work assumptions
- Describe how we may automatically generate Chaos experiments from models
  - observational models
  - models based on static analysis
  - models built using domain expertise

References

- https://github.com/carlpulley/docker-compose-testkit/tree/scala-meetup-krakow-2017
- *Chaos Engineering* by A.Basiri, N.Behnam, R.de Rooij, L.Hochstein, L.Kosewski, J.Reynolds and C.Rosenthal
- http://principlesofchaos.org/
- *The Weird Machines in Proof-Carrying Code* by J.Vanegue
- *The Byzantine Generals Problem* by L.Lamport, R.Shostak and M.Pease
- *Impossibility of Distributed Consensus with One Faulty Process* by M.Fischer, N.Lynch and M.Paterson
- *Freer monads, more extensible effects* by O.Kiselyov and H.Ishii

CAKESOLUTIONS

@cakesolutions

0845 617 1200

@

enquiries@cakesolutions.net