CAKESOLUTIONS
ENTERPRISE SOFTWARE SERVICES

MANCHESTER LONDON NEW YORK

Chaos Engineering and Runtime Monitoring of Distributed Reactive Systems

Dr. Carl Pulley
Cake Solutions @cakesolutions

The Goal

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

What Happens Next?

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

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

- 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!

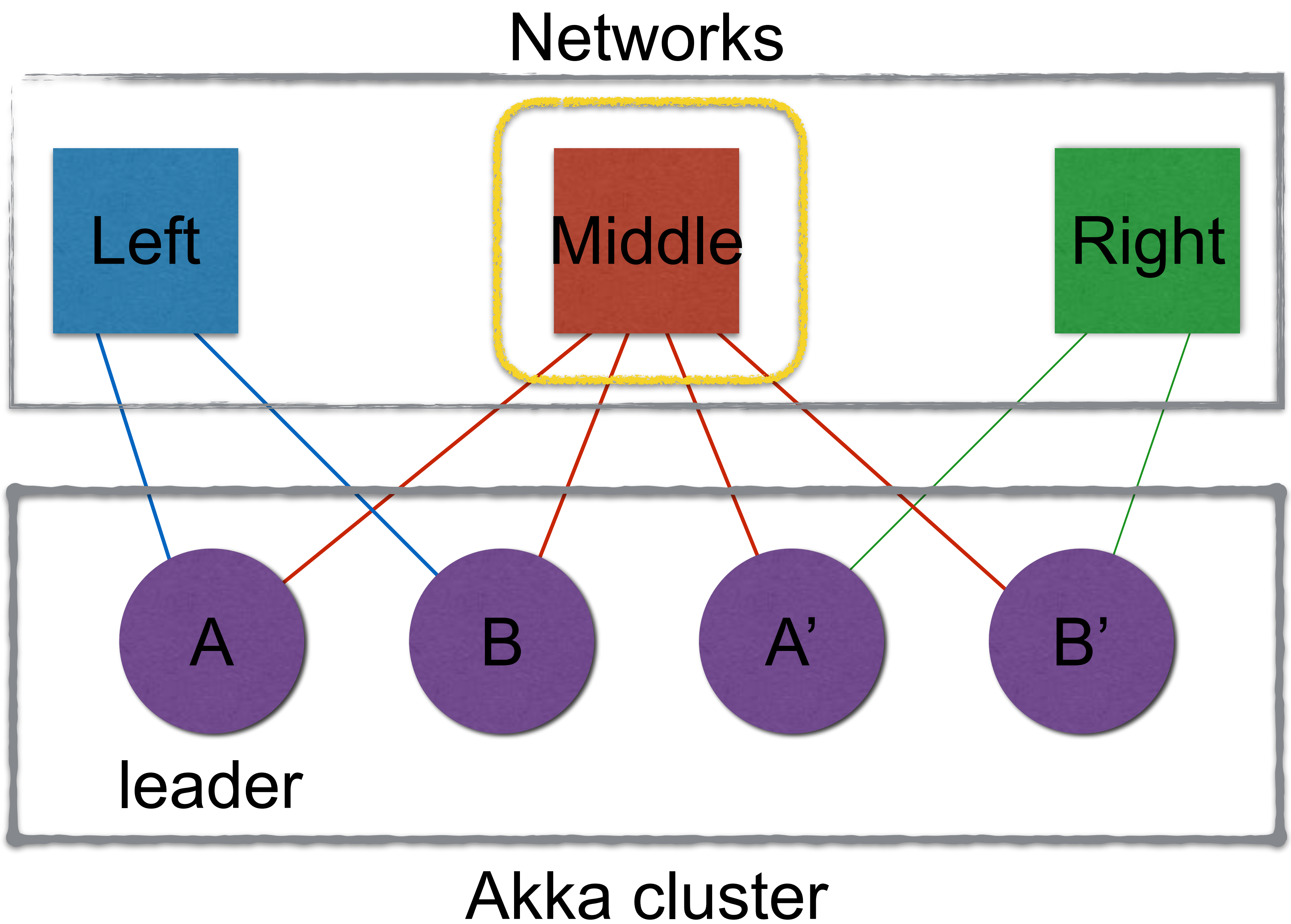
docker-compose-testkit

- 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

Demo Recipe

- (optional) describe application containers and networking
- (partially optional) define system instrumentation
  - view system as a black box
  - at the very least, need to define a layer of interaction code here!
- define the Chaos experiment
  - relate observed behaviours to data injection
- define runtime monitors (c.f. behaviours)
  - bind monitors to instrumented interface

Demo Overview



Auto-downing Enabled!!

Demo Overview

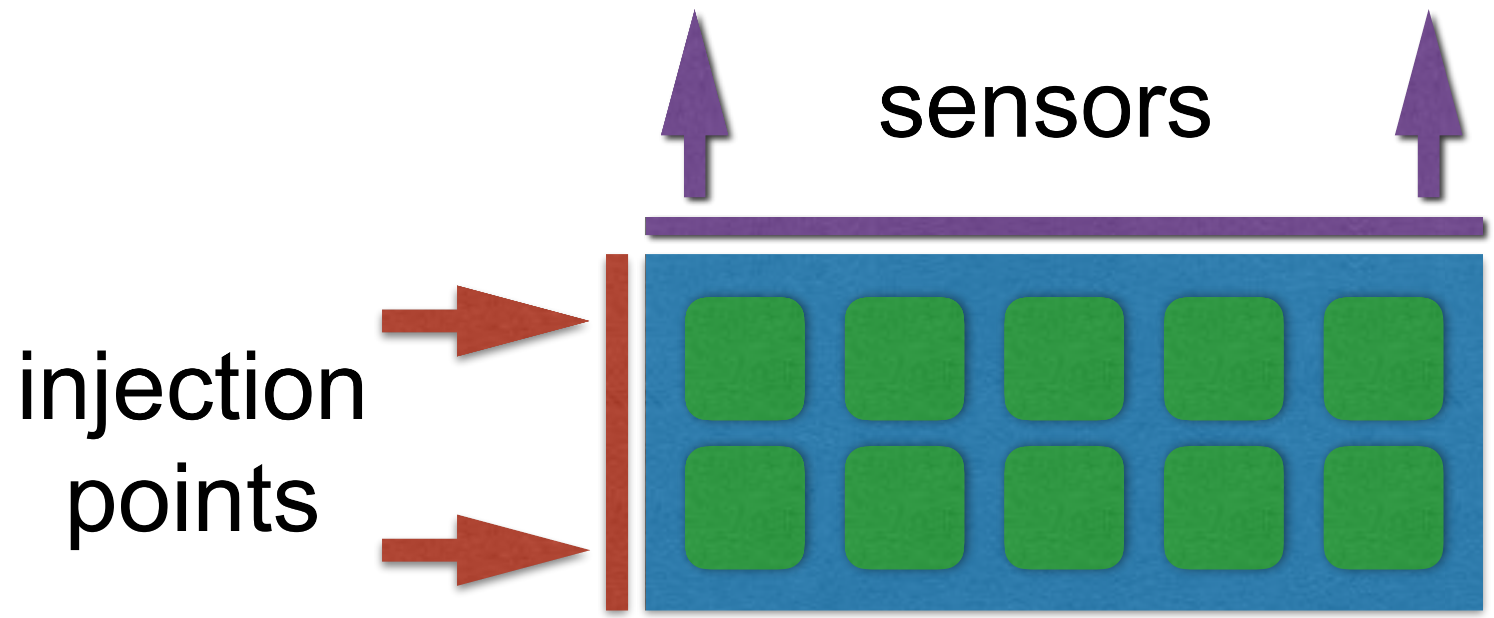
```
def clusterNode(name: String, network1: String, network2: String): String =
  s"""$name:
     |    template:
     |      resources:
     |        - cakesolutions.docker.jmx.akka
     |        - cakesolutions.docker.network.default.linux
     |      image: docker-compose-testkit-tests:$version
     |    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

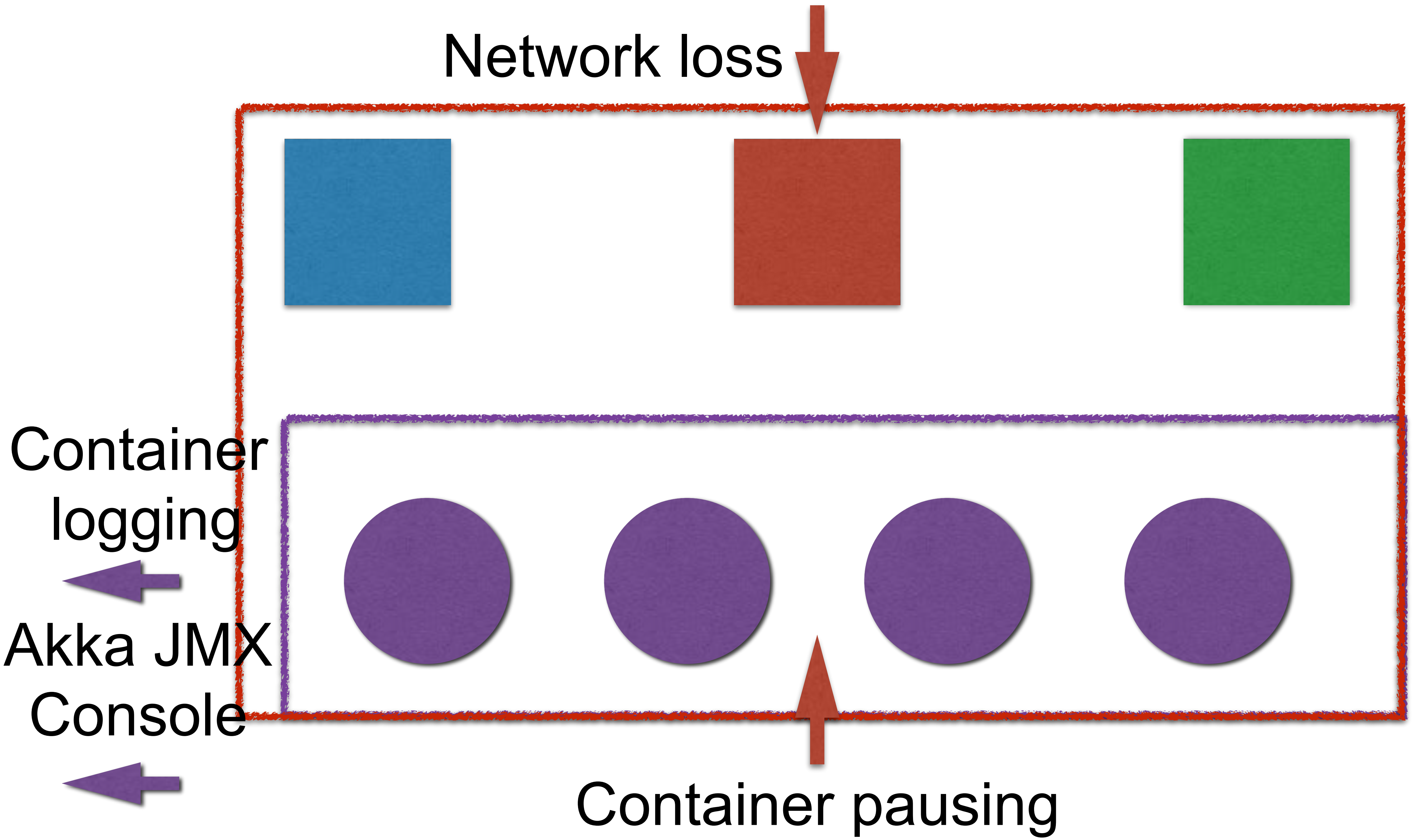
CAKESOLUTIONS

System Instrumentation



- define sensors
  - emit events of a fixed type
  - implemented using Monix Observables
- define injection points
  - allow typed data to influence the application
  - implemented using Monix Observers

Demo Overview



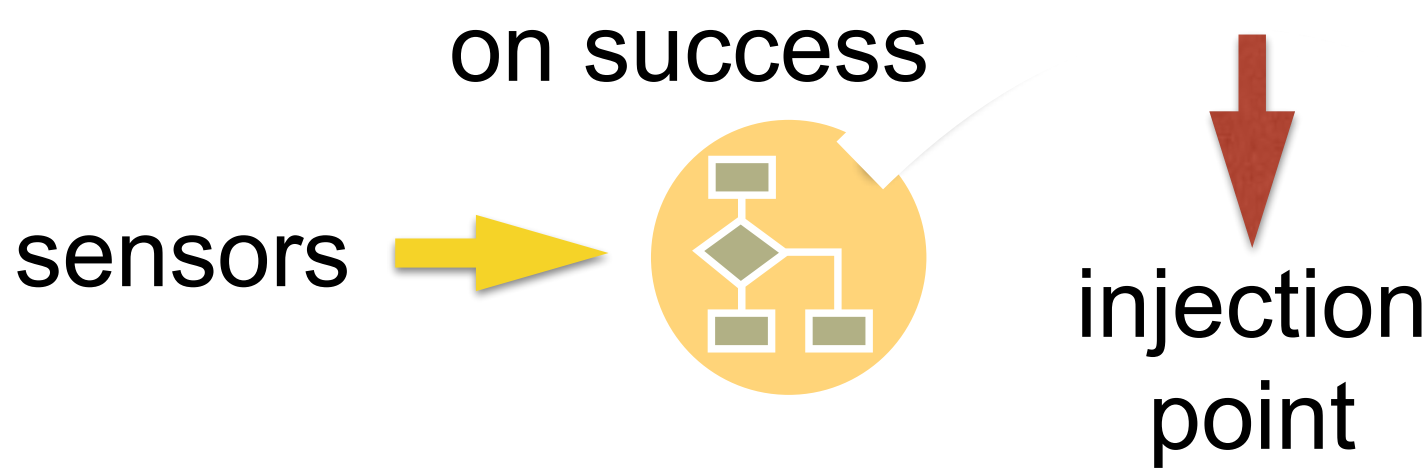
Demo Overview

```scala
final case class AkkaNodeSensors(
  log: TimedObservable.hot[LogEvent],
  members: TimedObservable.cold[AkkaClusterState]
)

object AkkaSensors {
  def apply(image: DockerImage)(implicit scheduler: Scheduler): AkkaNodeSensors = {
    AkkaNodeSensors(
      TimedObservable.hot(image.logging().publish),
      TimedObservable.cold(image.members())
    )
  }
}
```

For this presentation, injection points will be implemented using side effecting code

Chaos Experiment



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

Demo Overview

```
val testSimulation = for {
  _ <- ???
  _ = note("cluster formed, is stable and A an available leader")
  _ = rightNodeA.pause()
  _ = note("A' JVM GC pause starts")
  _ <- ???
  _ = note("A' is unreachable")
  _ = rightNodeA.unpause()
  _ = note("A' JVM GC pause ends")
  _ <- ???
  _ = note("cluster stabilised with A' as a member")
  _ = compose.network("middle").impair(Loss("100%"))
  _ = note("partition into left and right networks")
  _ <- ???
  _ = note("cluster split brains into 3 clusters: A & B; A'; B'")
} yield Accept()


testSimulation should observe(Accept())
```

Behavioural Property

- State machine
  - sensor events (i.e. Observable) cause machine to change state
  - (potentially) limit how long we may *linger* in a particular state
  - *StateTimeout* is observed when we linger in a given state for too long
  - may generate notification events
- Definition consists in defining a series of nested partial functions

Demo Overview

```scala
val available =
  MatchingAutomata[WaitToBeAvailable.type, Boolean](WaitToBeAvailable) {
  case _ => {
    case true =>
      Stop(Accept())
  }
}


val leader =
  MatchingAutomata[WaitForLeaderElection.type, Address](WaitForLeaderElection) {
  case _ => {
    case addr: Address if addr.host.contains("A") =>
      Stop(Accept())
  }
}
```

Demo Overview

```scala
val stableCluster =
  MatchingAutomata[StableCluster.type, List[Address]](StableCluster, 3.seconds) {
  case _ => {
    case addrs: List[Address @unchecked] if addrs.nonEmpty =>
      Stop(Fail(s"Detected $addrs as unreachable"))
    case StateTimeout =>
      Stop(Accept())
  }
}
```

Demo Overview

```
def clusterMembers(nodes: String*) =
  MatchingAutomata[ClusterMemberCheck.type, AkkaClusterState](ClusterMemberCheck) {
  case _ => {
    case AkkaClusterState(_, members, unreachable)
      if unreachable.isEmpty
         && members.filter(_.status == Up).flatMap(_.address.host) == Set(nodes: _*) =>
      Stop(Accept())
  }
}



clusterMembers("A", "B", "A'", “B'").run(clusterSensors("A").members)
```

```
available.run(clusterSensors("A").log) && leader.run(clusterSensors("A").log)
```

- Behavioural properties are executed when they bind to 1 or more *physical* sensor streams (i.e. Observables) and injection points (i.e. Observers)
- May combine these using propositional connectives

Demo Time!

Conclusion

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

References

- https://github.com/carlpulley/ docker-compose-testkit /tree/ scala-exchange-2016
- *Chaos Engineering* by Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, Casey Rosenthal
- http://principlesofchaos.org/
- *The Weird Machines in Proof-Carrying Code* by Julien 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

CAKESOLUTIONS

@cakesolutions

0845 617 1200

@

enquiries@cakesolutions.net