MANCHESTER LONDON NEW YORK

Using Chaos Engineering to Build Resilient Distributed Applications

Dr. Carl Pulley
Cake Solutions @cakesolutions

CAKESOLUTIONS

Chaos Engineering

Want to explore how these same techniques might be used to improve code development!

Chaos Engineering

- Emphasises an empirical approach to testing and monitoring distributed systems

Want to explore how these same techniques might be used to improve code development!

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 or business metrics)
  - randomly inject faults (e.g. by killing containers or changing networking behaviour)

Want to explore how these same techniques might be used to improve code development!

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 or business metrics)
  - randomly inject faults (e.g. by killing containers or changing networking behaviour)
- Infer weaknesses using deviations from expected or steady-state behaviour

Want to explore how these same techniques might be used to improve code development!

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 or business 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!

CAKESOLUTIONS

Want to explore how these same techniques might be used to improve code development!

Cloud Verification Problem

Cloud Verification Problem

- Verification is *"easy"* when we deploy into a static or constrained environment

Cloud Verification Problem

- Verification is *"easy"* when we deploy into a static or constrained environment
- Cloud deployment involves dynamic environments

Cloud Verification Problem

- Verification is *"easy"* when we deploy into a static or constrained environment
- Cloud deployment involves dynamic environments
  - we have little or no real control over these environments!

Cloud Verification Problem

- Verification is *"easy"* when we deploy into a static or constrained environment
- Cloud deployment involves dynamic environments
  - we have little or no real control over these environments!
  - deployment typically uses unverified processes

Cloud Verification Problem

- Verification is *"easy"* when we deploy into a static or constrained environment
- Cloud deployment involves dynamic environments
  - we have little or no real control over these environments!
  - deployment typically uses unverified processes
- Potential for unanticipated failure is highly likely

Cloud Verification Problem

- Verification is *"easy"* when we deploy into a static or constrained environment
- Cloud deployment involves dynamic environments
  - we have little or no real control over these environments!
  - deployment typically uses unverified processes
- Potential for unanticipated failure is highly likely
  - it doesn't matter how the code was produced!

Cloud Verification Problem

- Verification is *"easy"* when we deploy into a static or constrained environment
- Cloud deployment involves dynamic environments
  - we have little or no real control over these environments!
  - deployment typically uses unverified processes
- Potential for unanticipated failure is highly likely
  - it doesn't matter how the code was produced!
  - 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

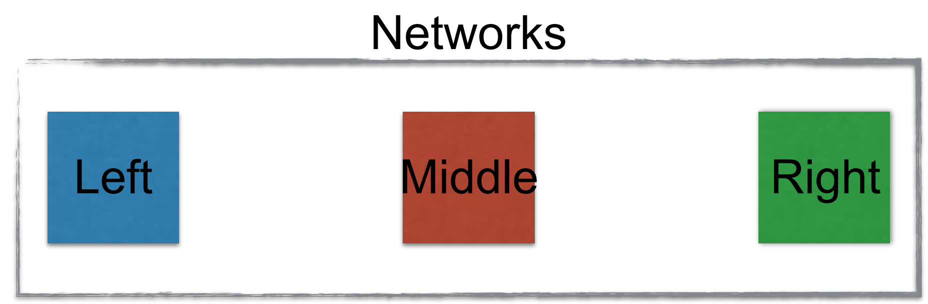
templates

templates

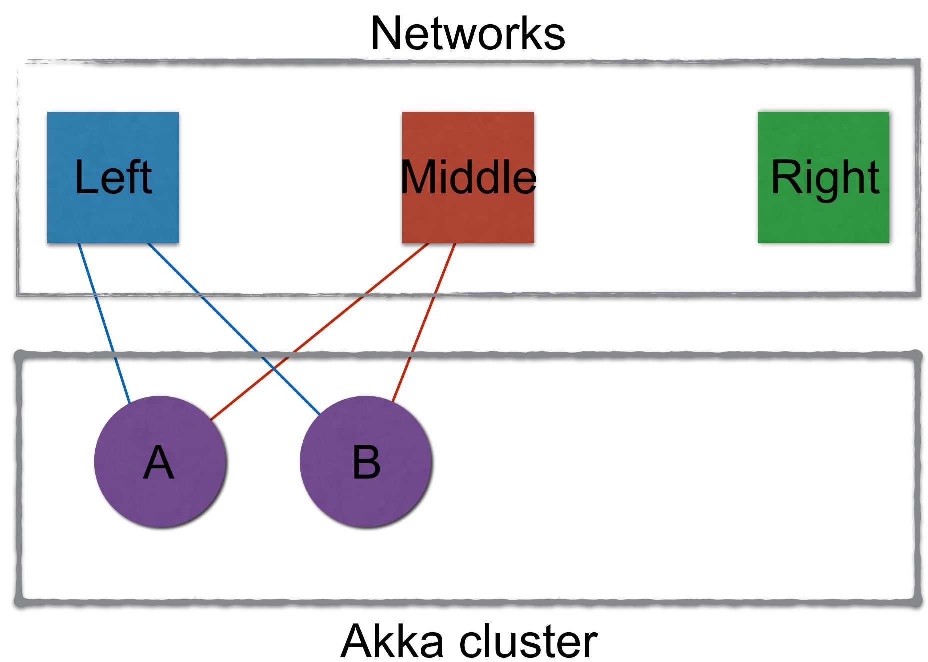
- Specify Docker image instrumentation layers
  - injects code and resources into image
- Specify library code for using instrumentation
  - code needs to use side effects
  - maintain a functional core using extensible effects (i.e. Eff)
  - sensor events represented using Monix Observables

Demo Overview

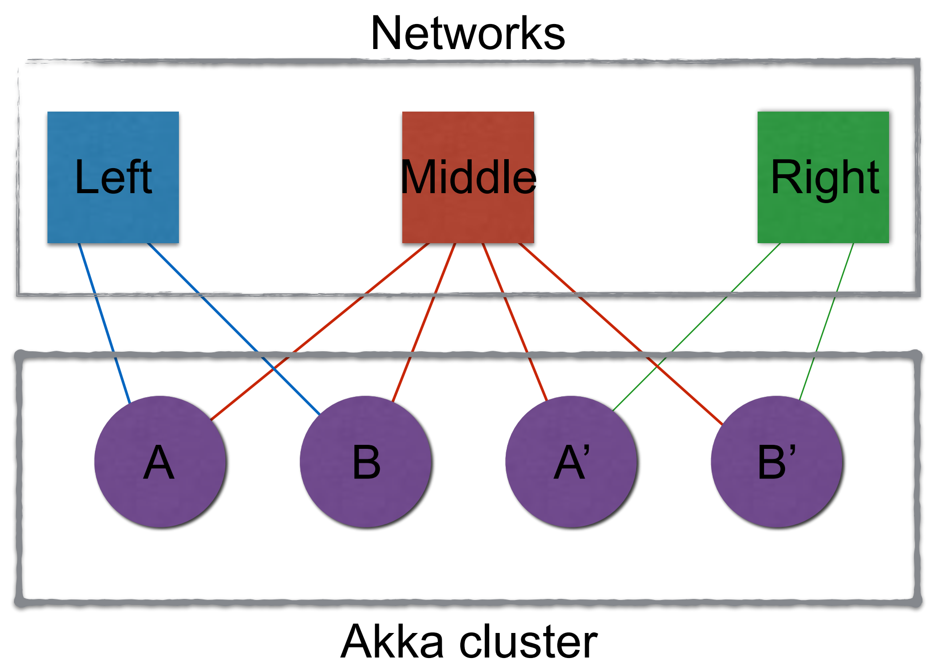
Demo Overview



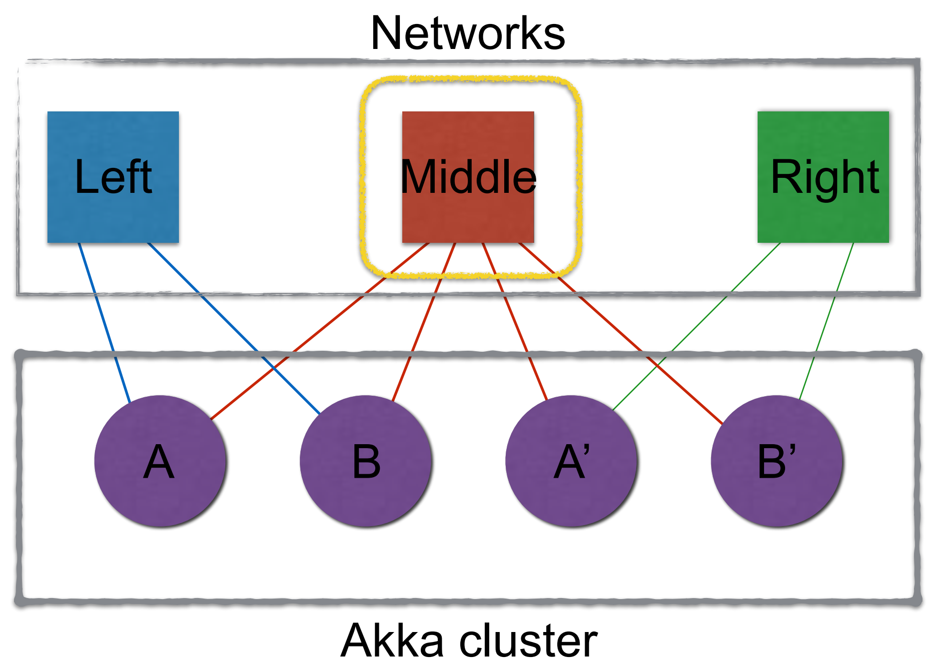
Demo Overview



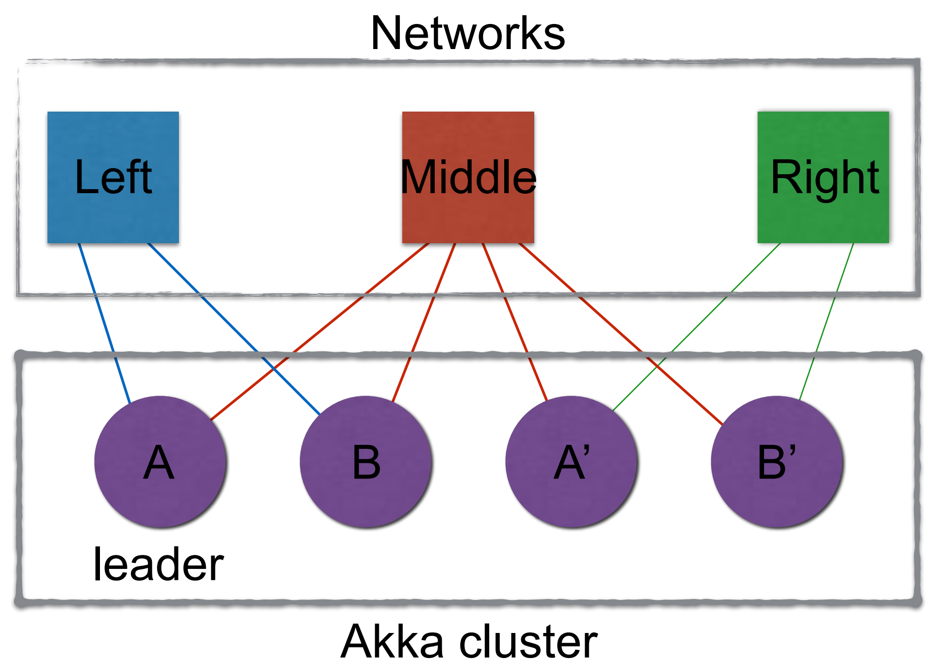
Demo Overview



Demo Overview



Demo Overview



Auto-downing Enabled!!

Demo Overview

```
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
)
```

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
)
```

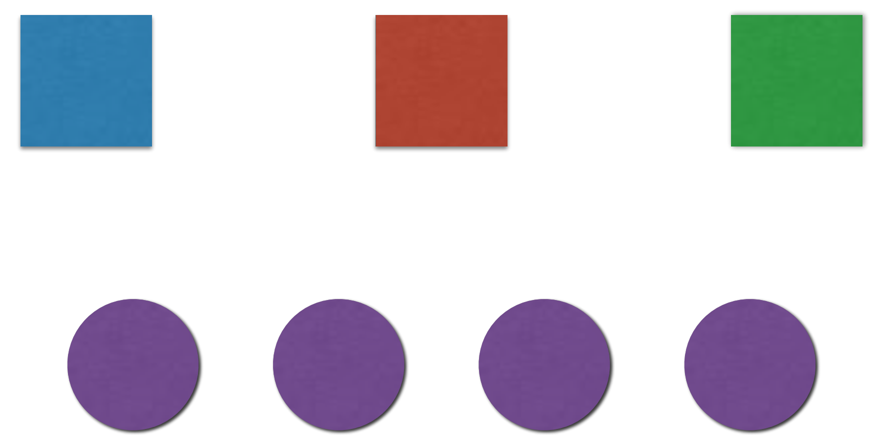
Demo Overview

```
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
)
```

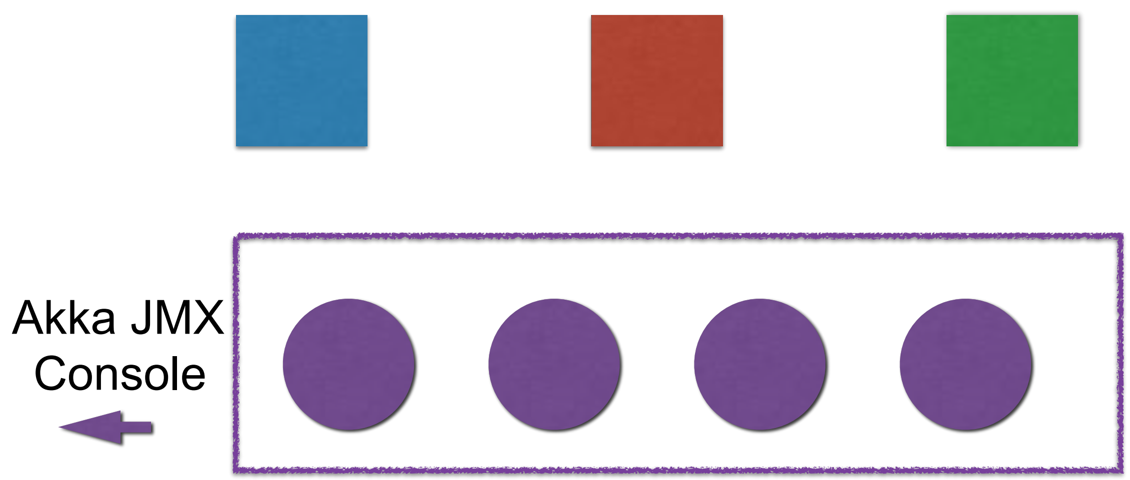
immutable container

Demo Overview



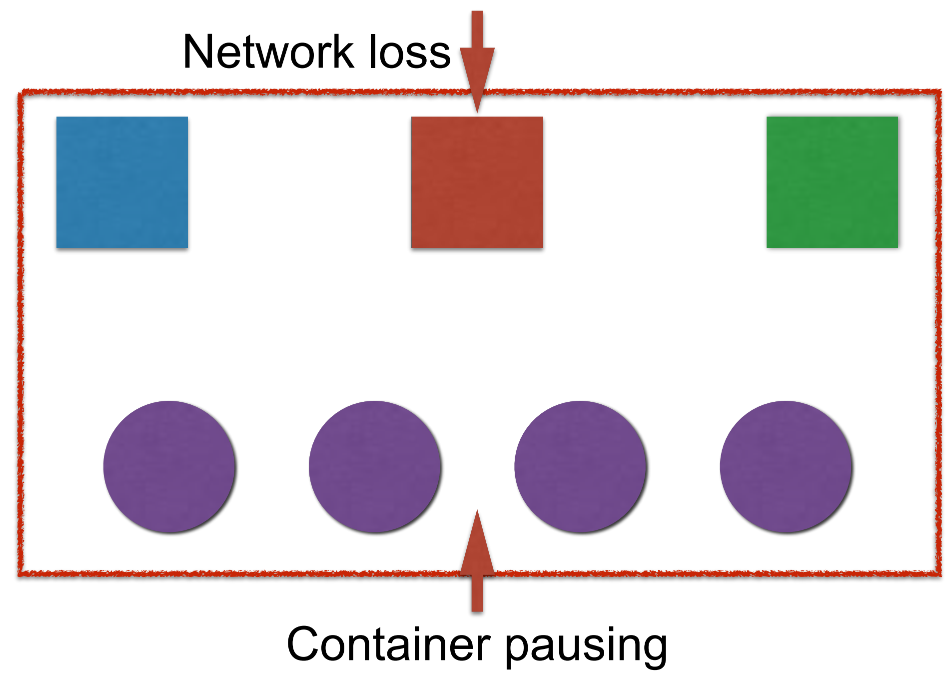
Point out our framework is capable of far more fault injection than this!
Necessary to perform some coding to define this instrumentation layer.
Docker-compose templating is used here!

Demo Overview



Point out our framework is capable of far more fault injection than this!
Necessary to perform some coding to define this instrumentation layer.
Docker-compose templating is used here!

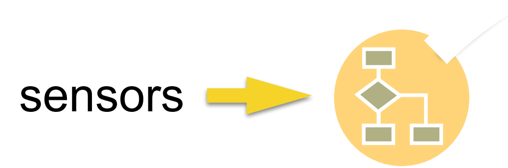
Demo Overview



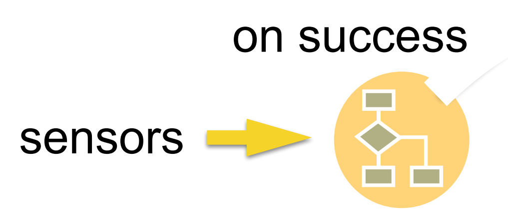
Point out our framework is capable of far more fault injection than this!
Necessary to perform some coding to define this instrumentation layer.
Docker-compose templating is used here!

Chaos Experiment

Chaos Experiment

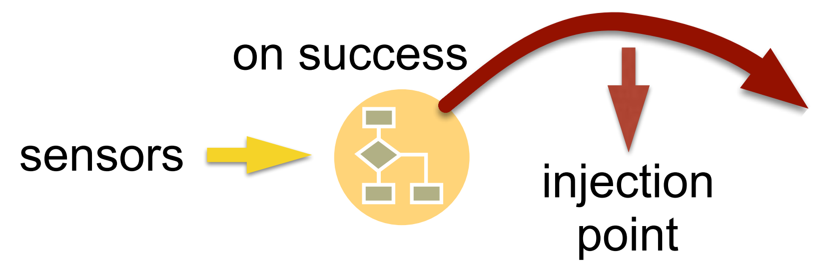


Chaos Experiment



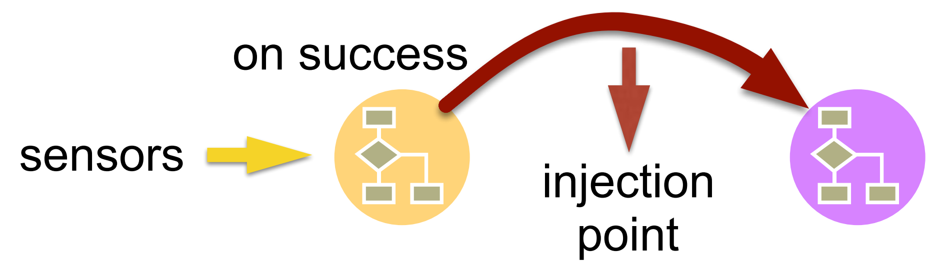
- Interpret this as follows:
  - when we successfully detect behaviour

Chaos Experiment



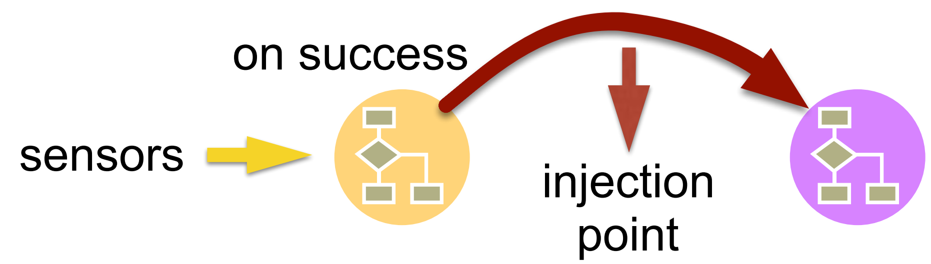
- Interpret this as follows:
  - when we successfully detect behaviour
  - action may happen

Chaos Experiment



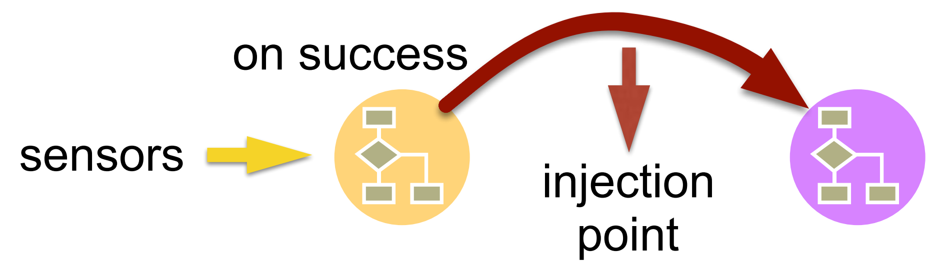
- Interpret this as follows:
  - when we successfully detect behaviour
  - action may happen
  - and we start monitoring for behaviour

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()
```

CAKESOLUTIONS

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

```
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

```scala
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) {
    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())
    }
  }
}
```

Initial State

Demo Overview

Observed Event

```scala
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

Demo Overview

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











  } yield Accept()
```

CAKESOLUTIONS

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

```scala
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

```
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…
```

CAKESOLUTIONS

Demo Overview

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

Demo Time!

docker events --filter type=container
sbt clean “dockerComposeTests/testOnly *AutoDownSplitBrainDockerTest”

Development Use Cases

Development Use Cases

- Can we use instrumentation to investigate *difficult to observe* issues?

Development Use Cases

- Can we use instrumentation to investigate *difficult to observe* issues?
- For example:
  - directly observe and extract networking evidence
  - observe impact of resource starved Docker containers
    - e.g. low memory or small thread pools

Development Use Case: Networking

Development Use Case: Networking

- Networking issues
  - instrument container with *iptables*
  - use runtime monitors to validate received/sent network traffic expectations
    - capture PCAP data as a side effect of monitoring for offline examination

Development Use Case: Container Resource Stress

Development Use Case: Container Resource Stress

- Use *LIB_PRELOAD* “trick” to load application code via *libfiu*

Development Use Case: Container Resource Stress

- Use *LIB_PRELOAD* “trick” to load application code via *libfiu*
- libfiu - Fault Injection in Userspace
  - DSL allows system calls to return with an error code
    - every time or randomly specified
  - reliably simulate container memory, CPU or thread resourcing issues
    - often very difficult to detect!

Development Use Case: JVM

Development Use Case: JVM

- Modify JVM class loaders to instrument runtime application code

Development Use Case: JVM

- Modify JVM class loaders to instrument runtime application code
- Byteman
  - rule based instrumentation DSL
    - event-condition-action (ECA) rules
  - operates at the the Java (not bytecode) level
  - enables principled experiments to guide investigation into difficult to observe issues
    - exceptions that get *swallowed*
    - validate expected data and control flows
    - etc.

Next Steps

Next Steps

- Failure control plane
  - sidecar failure control node per Docker container
  - failure agent within each container
  - lower Dockerfile impact

Next Steps

- Failure control plane
  - sidecar failure control node per Docker container
  - failure agent within each container
  - lower Dockerfile impact
- Describe how we may automatically generate Chaos experiments from models
  - observational models
  - models based on static analysis
  - models built using domain expertise

Conclusion

Conclusion

- Taken steady state behavioural specifications and developed a composable DSL for experiments
  - by relating data injection traces to observed system behaviour

Conclusion

- 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

Conclusion

- 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
- Shown how these techniques can be used within debugging scenarios

References

- https://github.com/carlpulley/ docker-compose-testkit /tree/ scala-sphere-2017
- *Chaos Engineering* by A.Basiri, N.Behnam, R.de Rooij, L.Hochstein, L.Kosewski, J.Reynolds and C.Rosenthal
- *Freer monads, more extensible effects* by O.Kiselyov and H.Ishii
- *eff* - https://github.com/atnos-org/eff
- *libfiu* - https://blitiri.com.ar/p/libfiu/
- *Byteman* - http://byteman.jboss.org/

CAKESOLUTIONS

@cakesolutions

0845 617 1200

enquiries@cakesolutions.net