Akka:

Simpler Concurrency, Scalability & Fault-tolerance through Actors

Jonas Bonér
Scalable Solutions
jonas@jonasboner.com
twitter: @jboner

The problem

It is way too hard to build:

1. correct highly concurrent systems
2. truly scalable systems
3. fault-tolerant systems that self-heals

...using "state-of-the-art" tools

Introducing akka

Vision

Simpler

- Concurrency
- Scalability
- Fault-tolerance

Vision

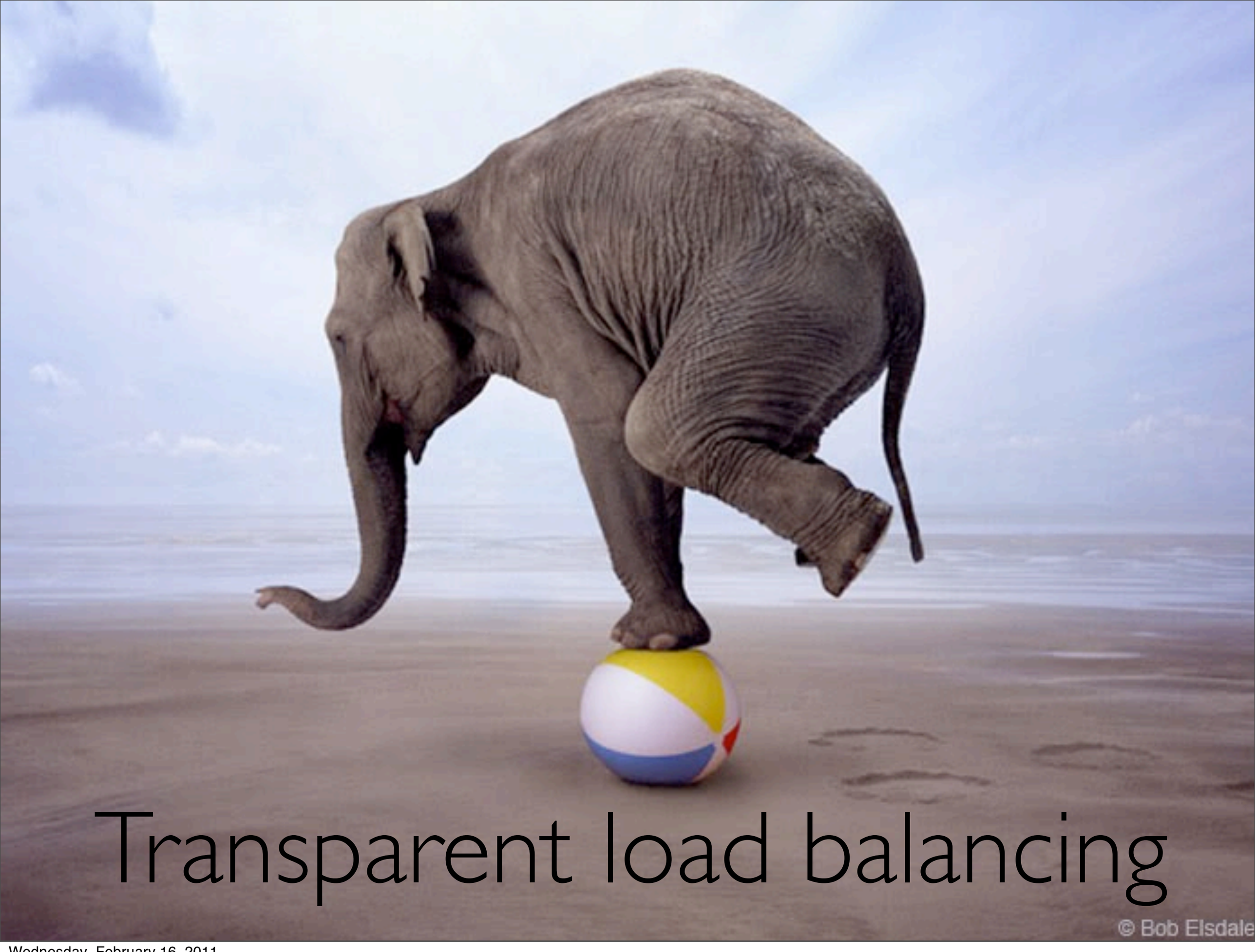
...with One single unified

- Programming model
- Runtime service

Manage system overload

Scale up & Scale out

Replicate and distribute for fault-tolerance



Transparent load balancing

Overview

Akka is the platform for the next generation event-driven, scalable and fault-tolerant architectures on the JVM

Simpler Concurrency

Event-driven Architecture

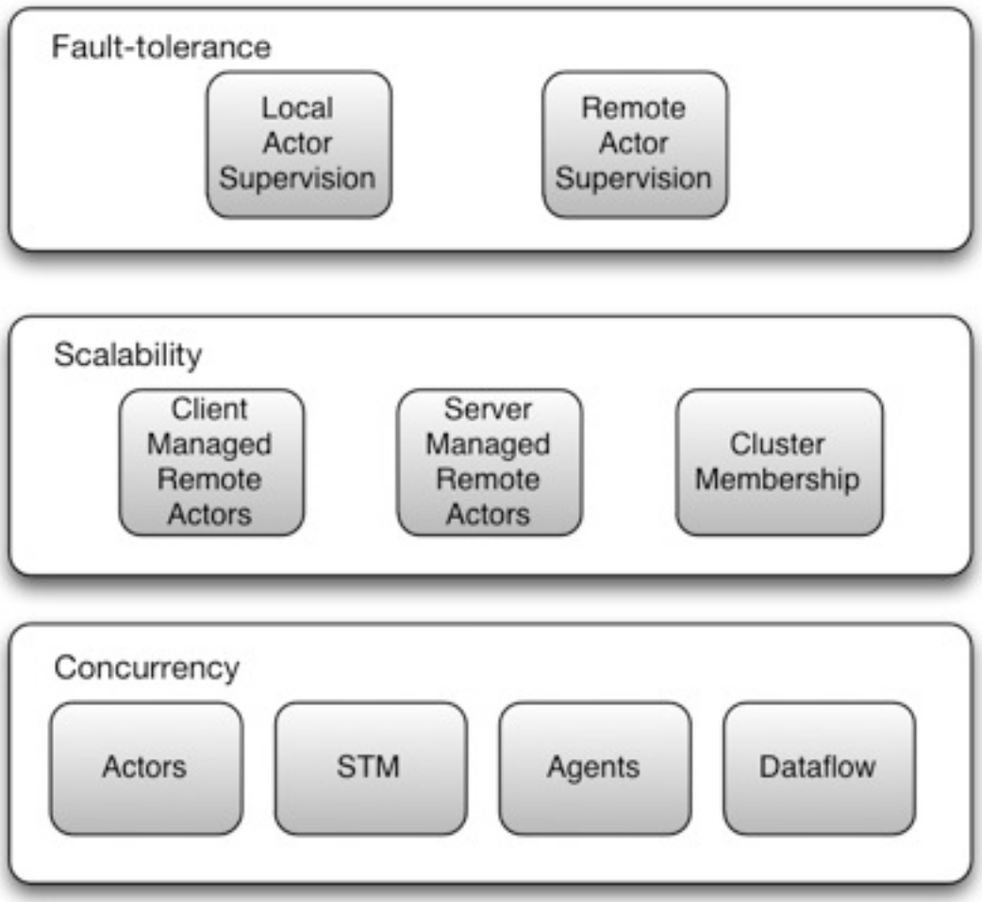
True Scalability

Fault-tolerance

Transparent Remoting

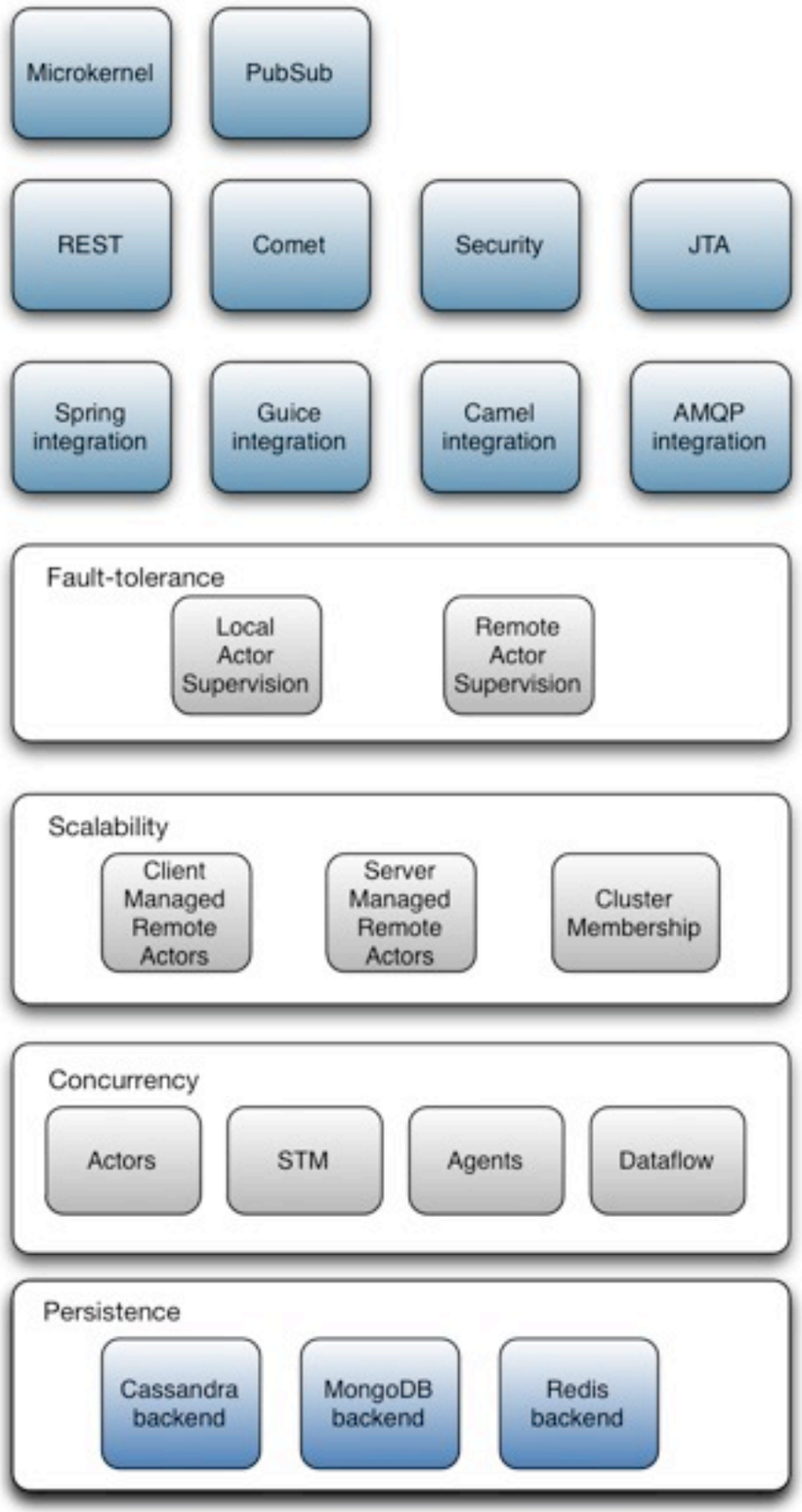
Java & Scala API

ARCHITECTURE



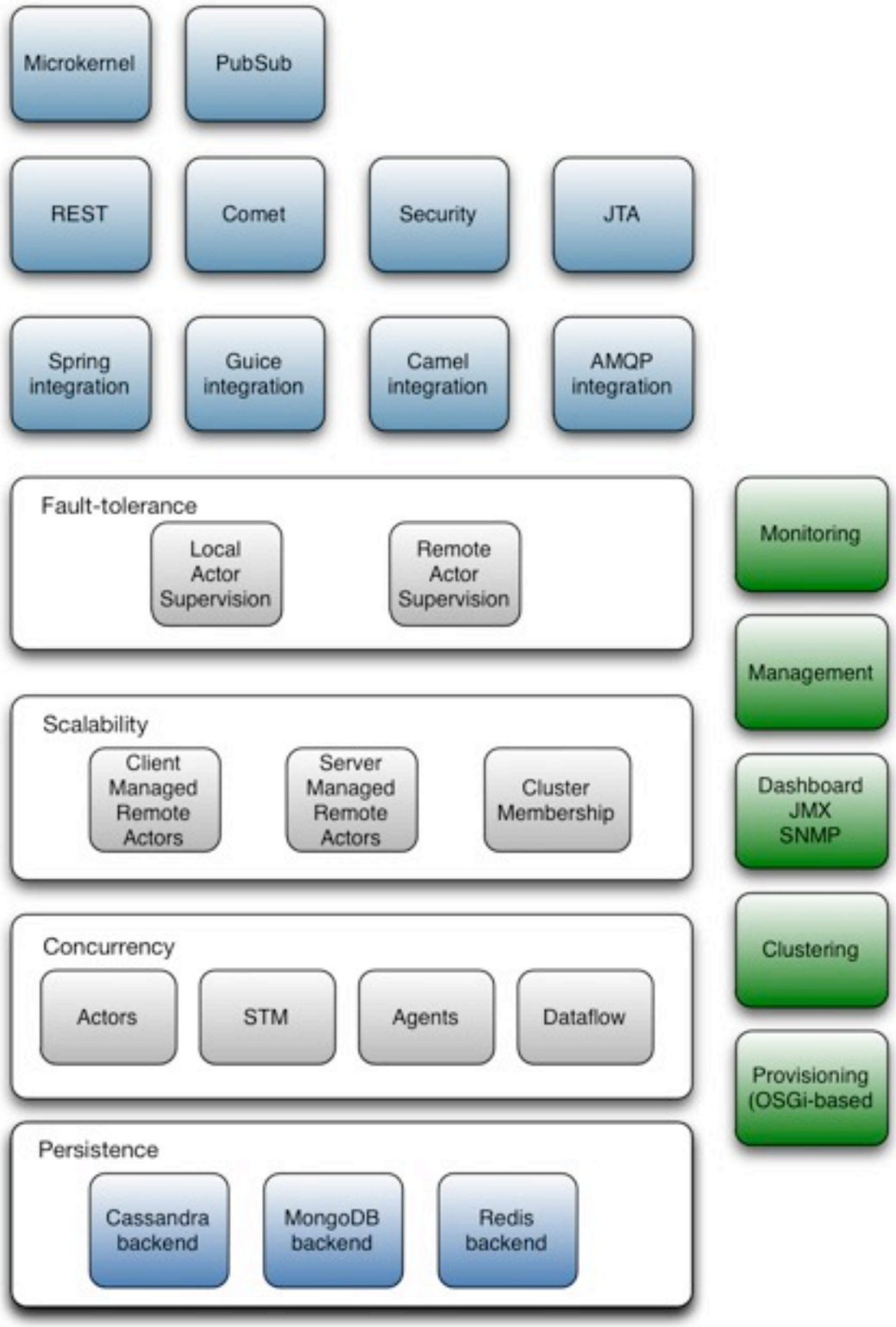
CORE SERVICES

ARCHITECTURE



ADD-ON MODULES

ARCHITECTURE



CLOUDY

AKKA

WHERE IS AKKA USED?

SOME EXAMPLES:

FINANCE

- Stock trend Analysis & Simulation
- Event-driven messaging systems

BETTING & GAMING

- Massive multiplayer online gaming
- High throughput and transactional betting

TELECOM

- Streaming media network gateways

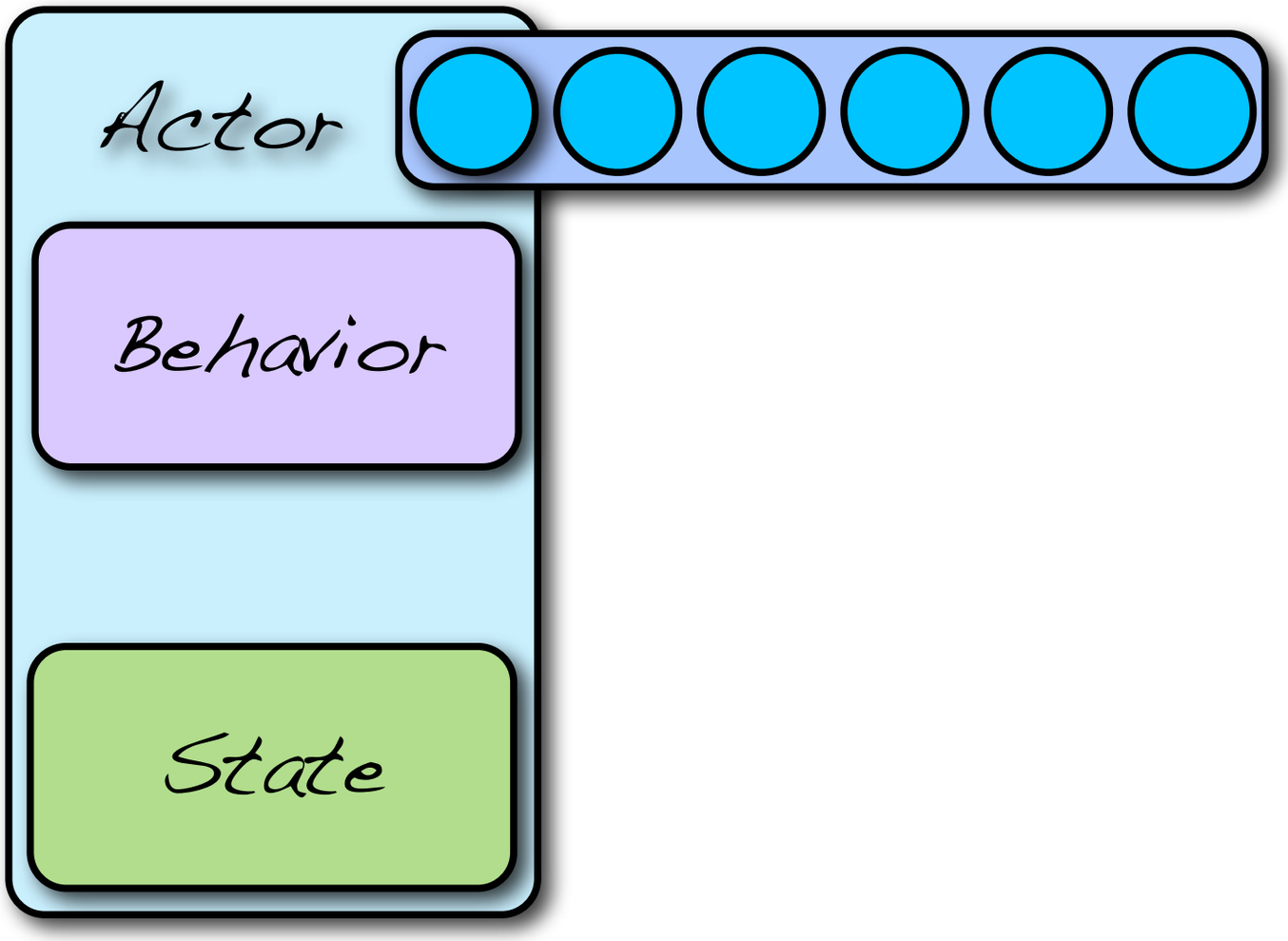
SIMULATION

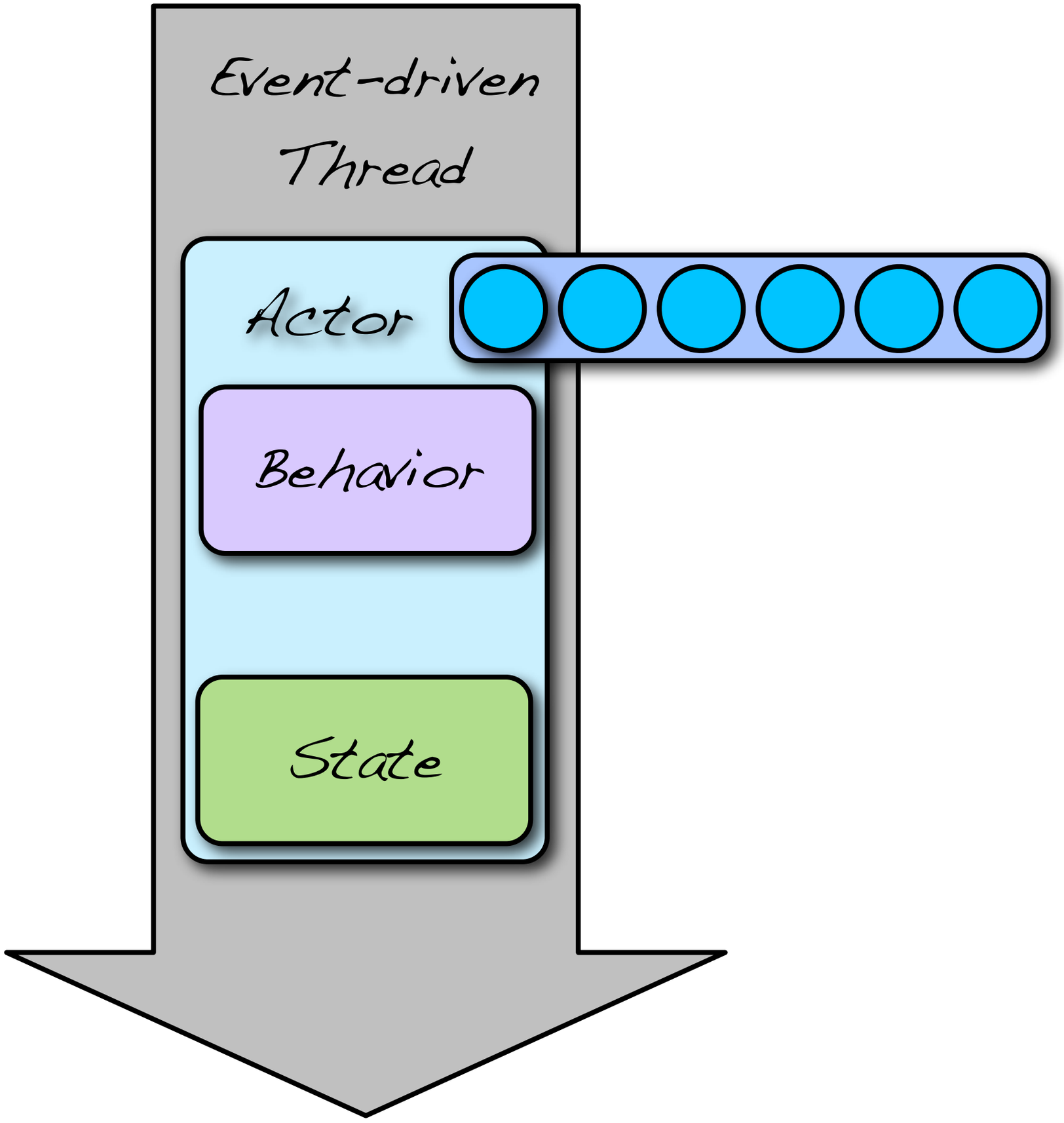
- 3D simulation engines

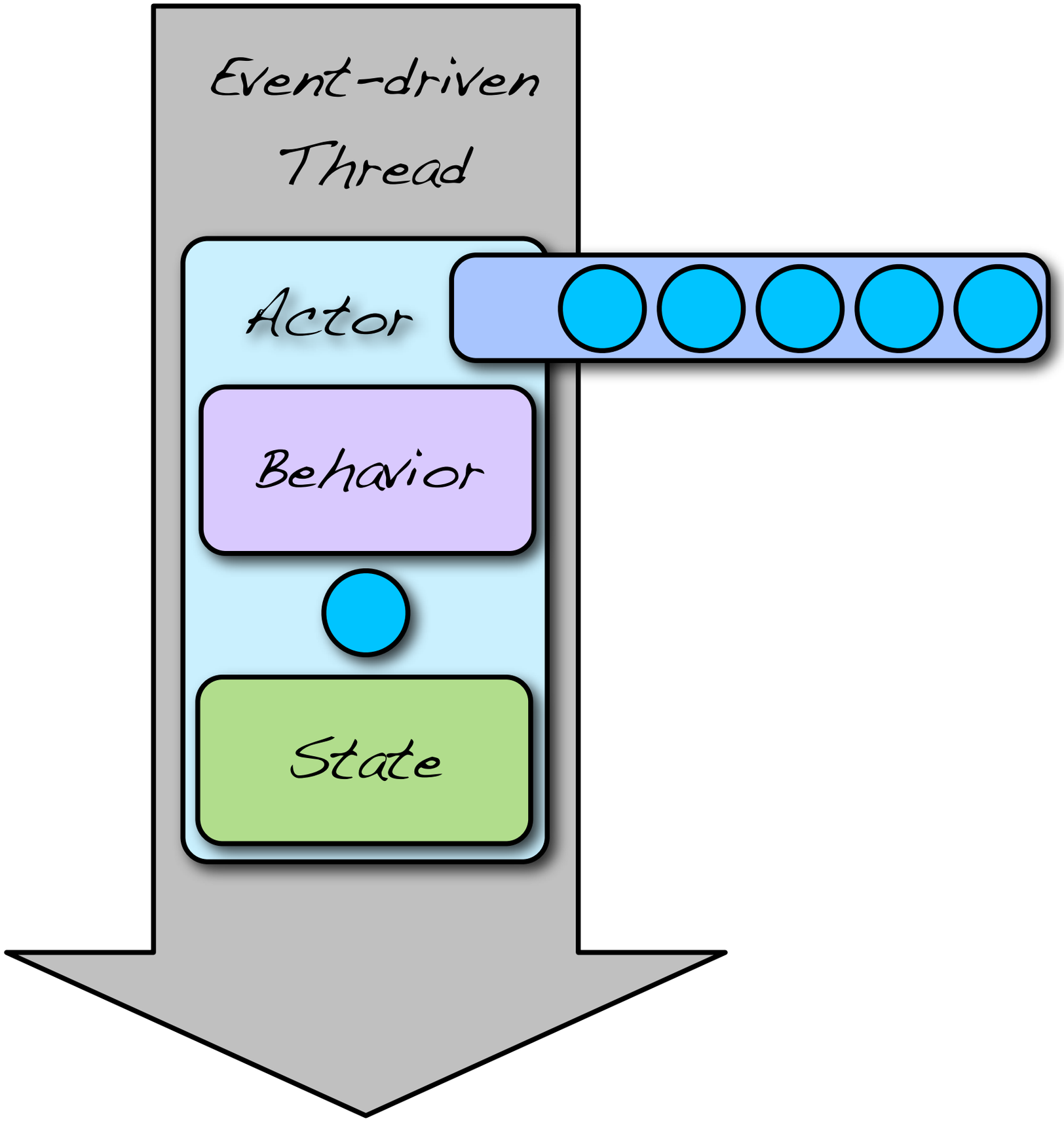
E-COMMERCE

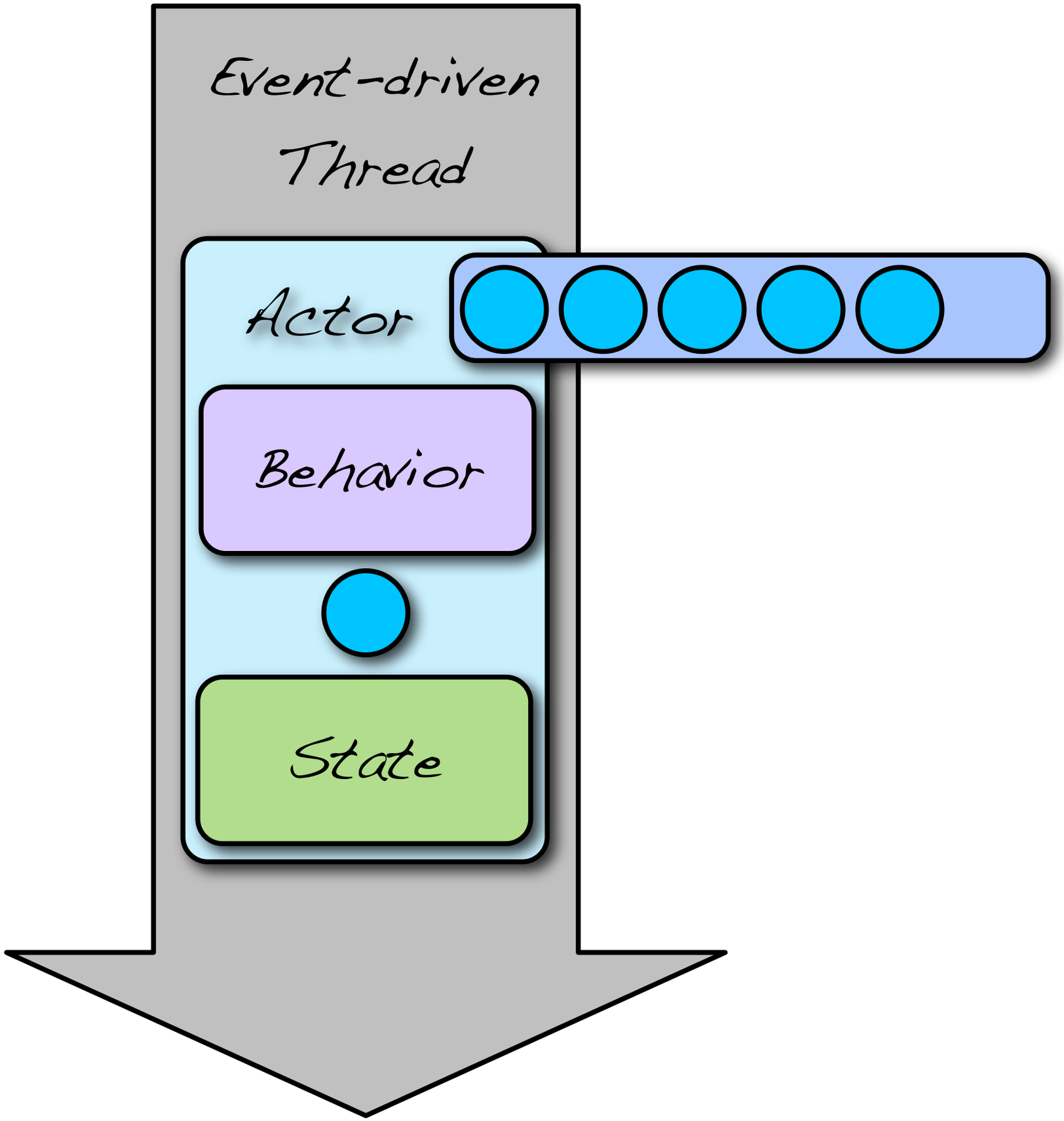
- Social media community sites

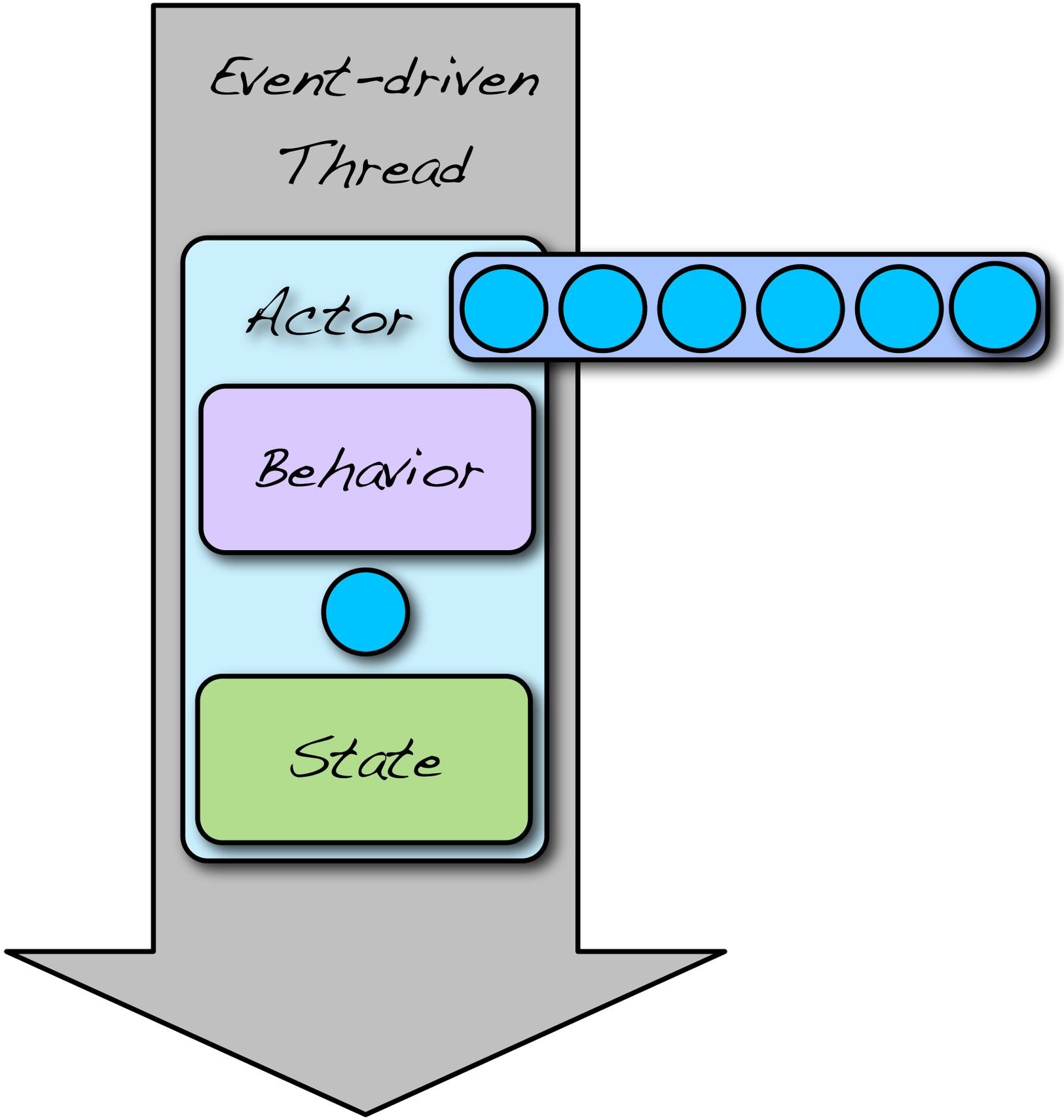
What is an Actor?

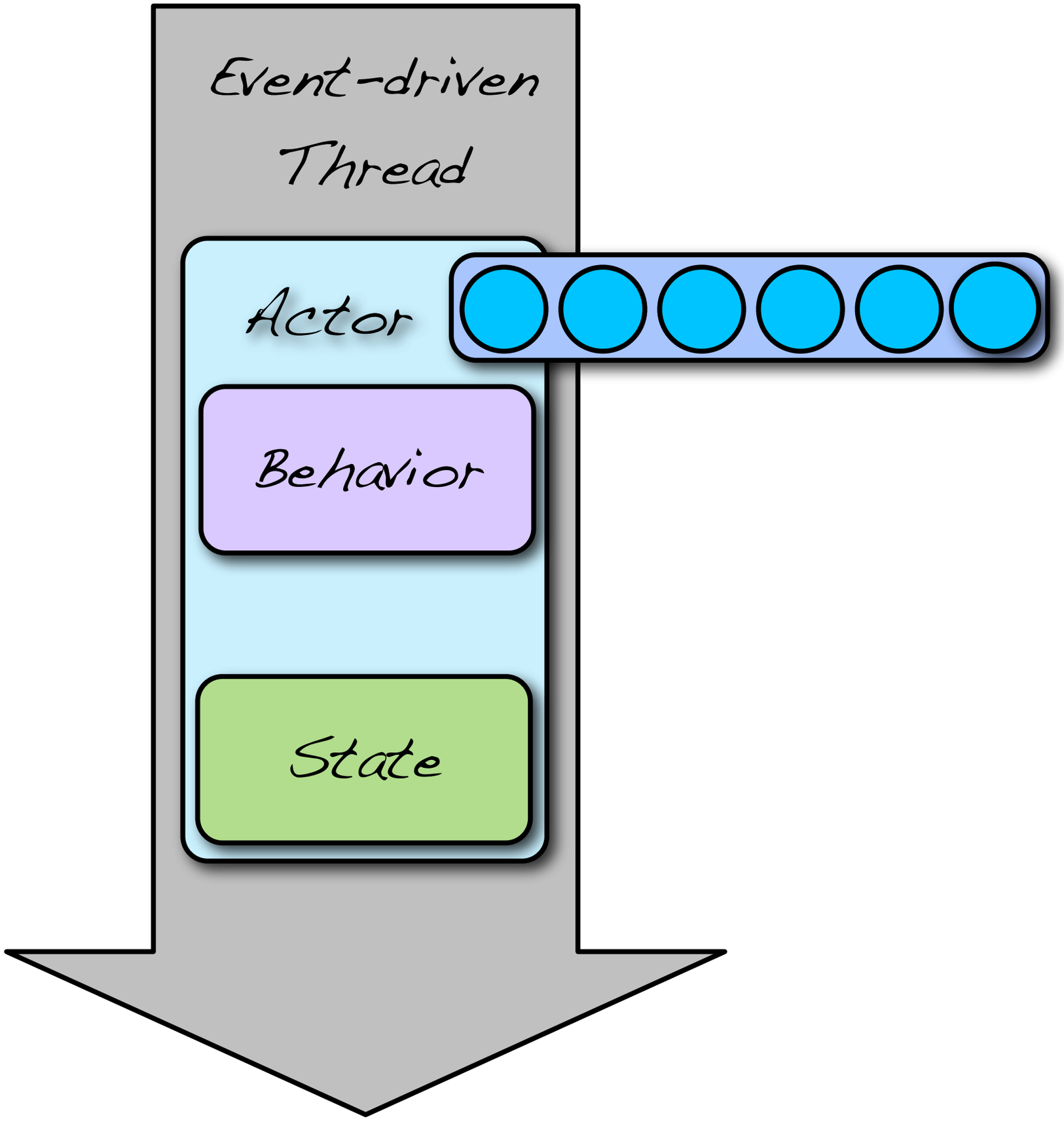

Actor
Behavior
State

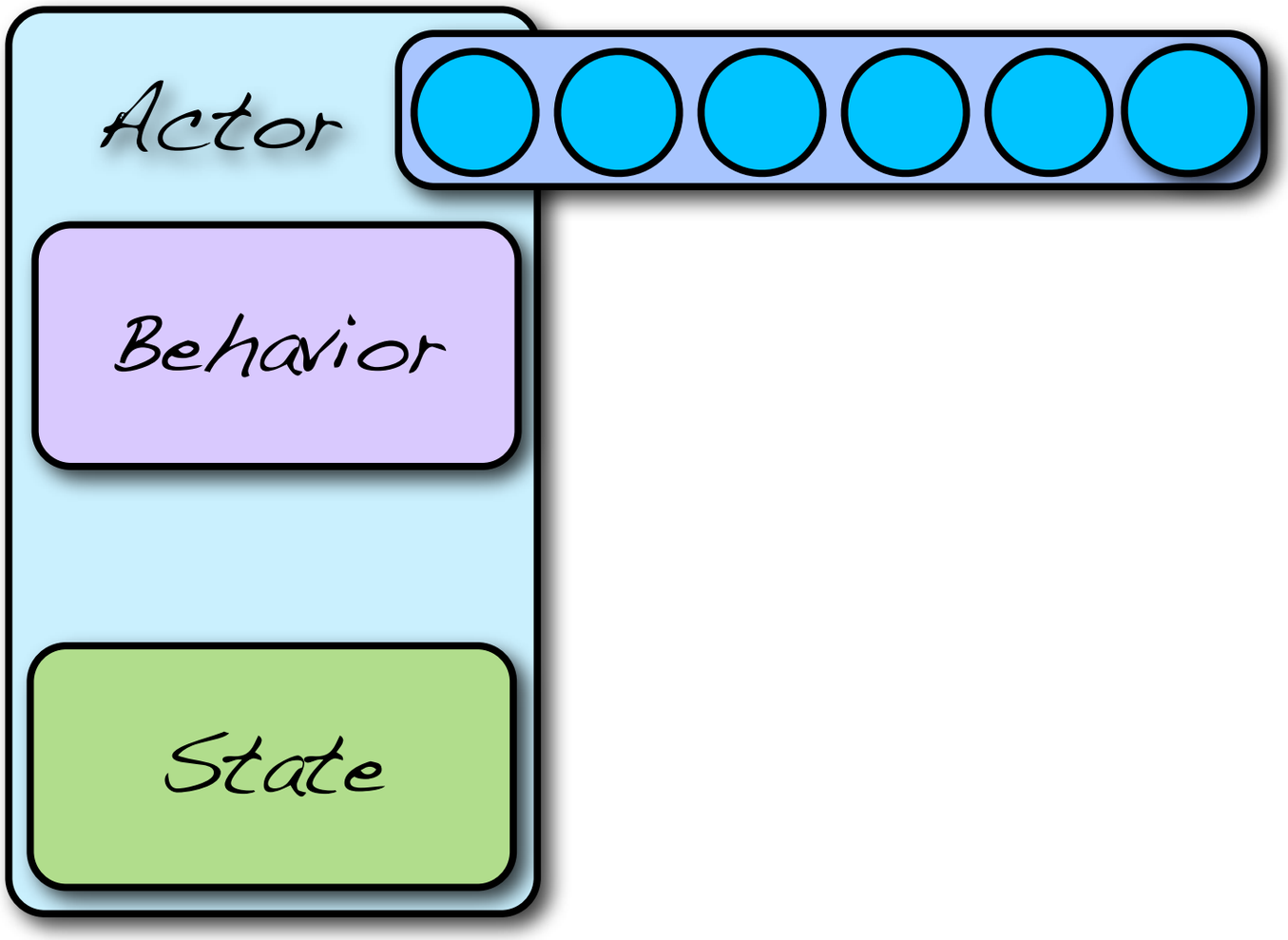

Event-driven
Thread
Actor
Behavior
State

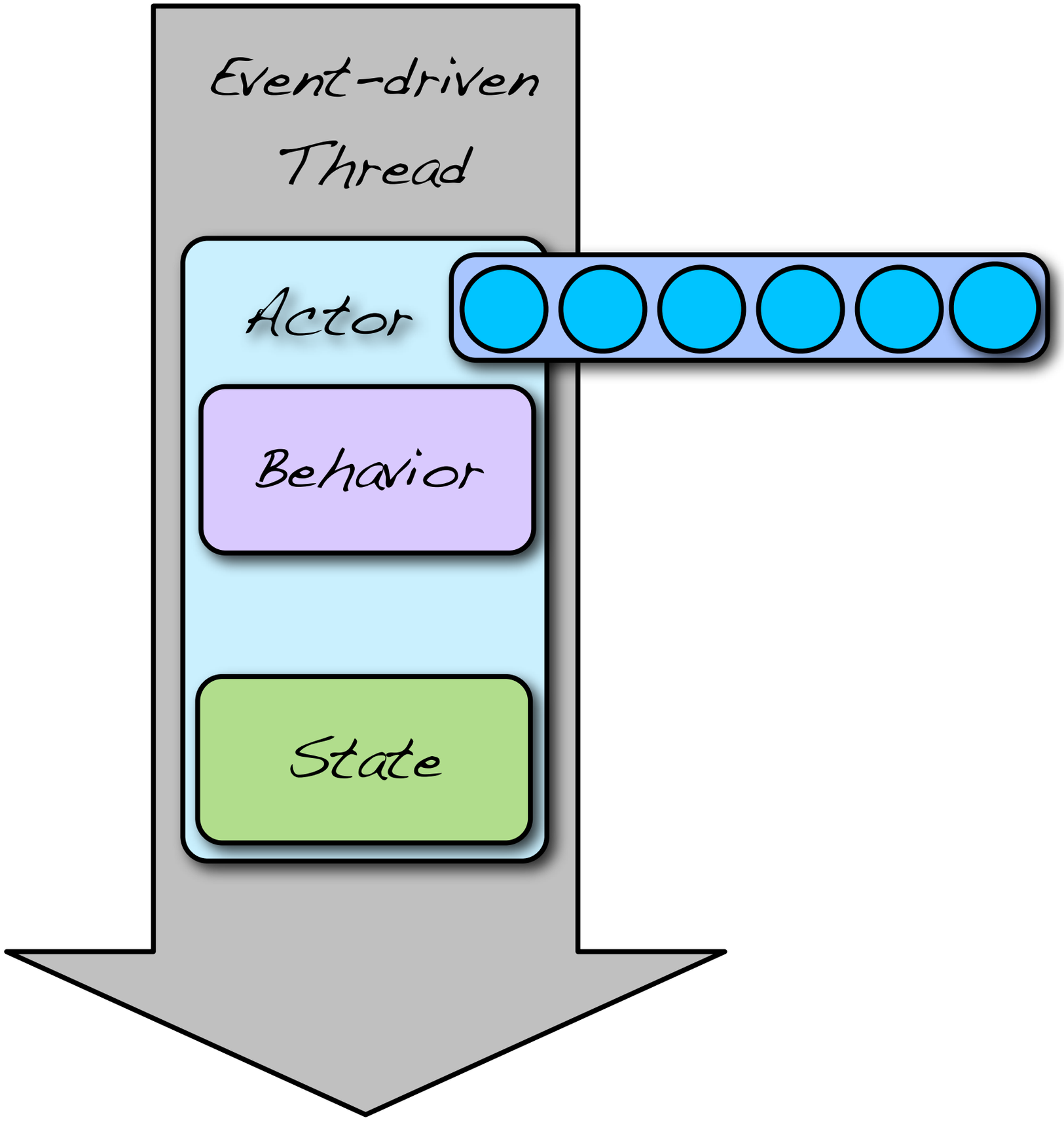

Event-driven
Thread
Actor
Behavior
State


Event-driven
Thread
Actor
Behavior
State


Event-driven
Thread
Actor
Behavior
State


Event-driven
Thread
Actor
Behavior
State


Actor
Behavior
State


Event-driven
Thread
Actor
Behavior
State

Akka Actors

one tool in the toolbox

Actor Model Benefits

- Easier to reason about
- Raised abstraction level
- Easier to avoid
  - Race conditions
  - Deadlocks
  - Starvation
  - Live locks

Actors

```
case object Tick

class Counter extends Actor {
  var counter = 0

  def receive = {
    case Tick =>
      counter += 1
      println(counter)
  }
}
```

Create Actors

```
val counter = actorOf[Counter]
```

`counter` is an `ActorRef`

Create Actors

```
val actor = actorOf(new MyActor(..))
```

create actor with constructor arguments

Start actors

```
val counter = actorOf[Counter]
counter.start
```

Start actors

```
val counter = actorOf[Counter].start
```

Stop actors

```
val counter = actorOf[Counter].start
counter.stop
```

life-cycle callbacks

```
class MyActor extends Actor {

  override def preStart = {
    ... // called before ‘start’
  }

  override def postStop = {
    ... // called after ‘stop’
  }
}
```

Send: !

```
counter ! Tick
```

fire-forget

Send: !!

```
val result = (actor !! Message).as[String]
```

uses Future under the hood (with time-out)

Send: !!!

```
// returns a future
val future = actor !!! Message
future.await
val result = future.get


...
Futures.awaitOne(List(fut1, fut2, ...))
Futures.awaitAll(List(fut1, fut2, ...))
```

returns the Future directly

Future

```
val future1, future2, future3 =
  new DefaultCompletableFuture(1000)

future1.await
future2.onComplete(f => ...)
future3.map((f) => ...)

future1.completeWithResult(...)
future2.completeWithException(...)
future3.completeWith(future2)
```

Reply

```
class SomeActor extends Actor {
  def receive = {
    case User(name) =>
      // use reply
      self.reply(“Hi ” + name)
  }
}
```

Reply

```
class SomeActor extends Actor {
  def receive = {
    case User(name) =>
      // store away the sender
      // to use later or
      // somewhere else
      ... = self.sender
  }
}
```

Reply

```
class SomeActor extends Actor {
  def receive = {
    case User(name) =>
      // store away the sender future
      // to resolve later or
      // somewhere else
      ... = self.senderFuture
  }
}
```

HotSwap

```
self become {
  // new body
  case NewMessage =>
    ...
}
```

HotSwap

```
actor ! HotSwap {
   // new body
   case NewMessage =>
      ...
}
```

HotSwap

```
self.unbecome()
```

ActorRegistry

```
val actors = Actor.registry.actors()
val actors = Actor.registry.actorsFor[TYPE]
val actors = Actor.registry.actorsFor(id)
val actor  = Actor.registry.actorFor(uuid)
Actor.registry.shutdownAll()
```

Set dispatcher

```
class MyActor extends Actor {
  self.dispatcher = Dispatchers
    .newThreadBasedDispatcher(self)

  ...
}

actor.dispatcher = dispatcher // before started
```

Let it crash

fault-tolerance

Stolen from

Erlang

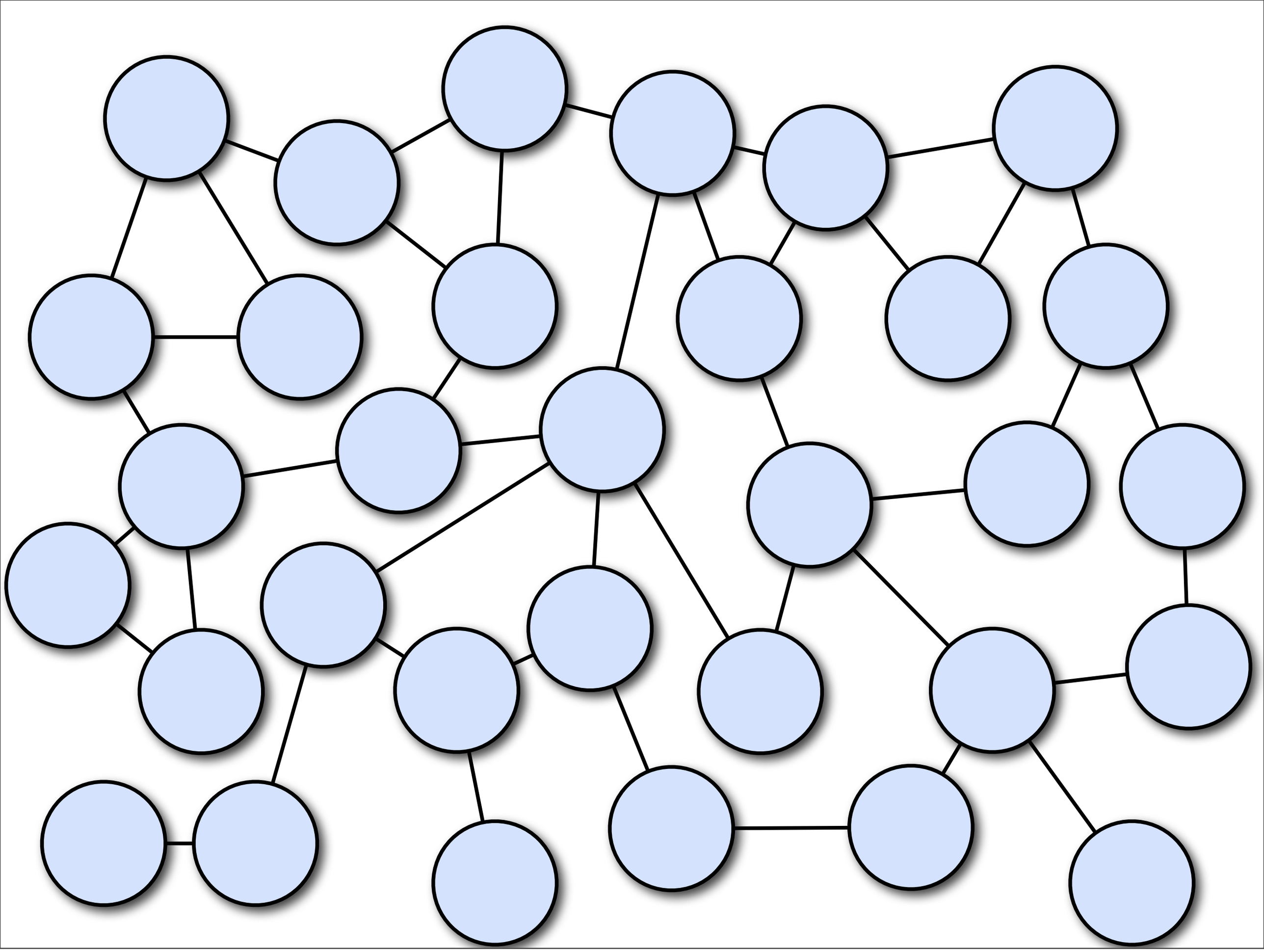
9

nines

...let's take a

standard OO

application

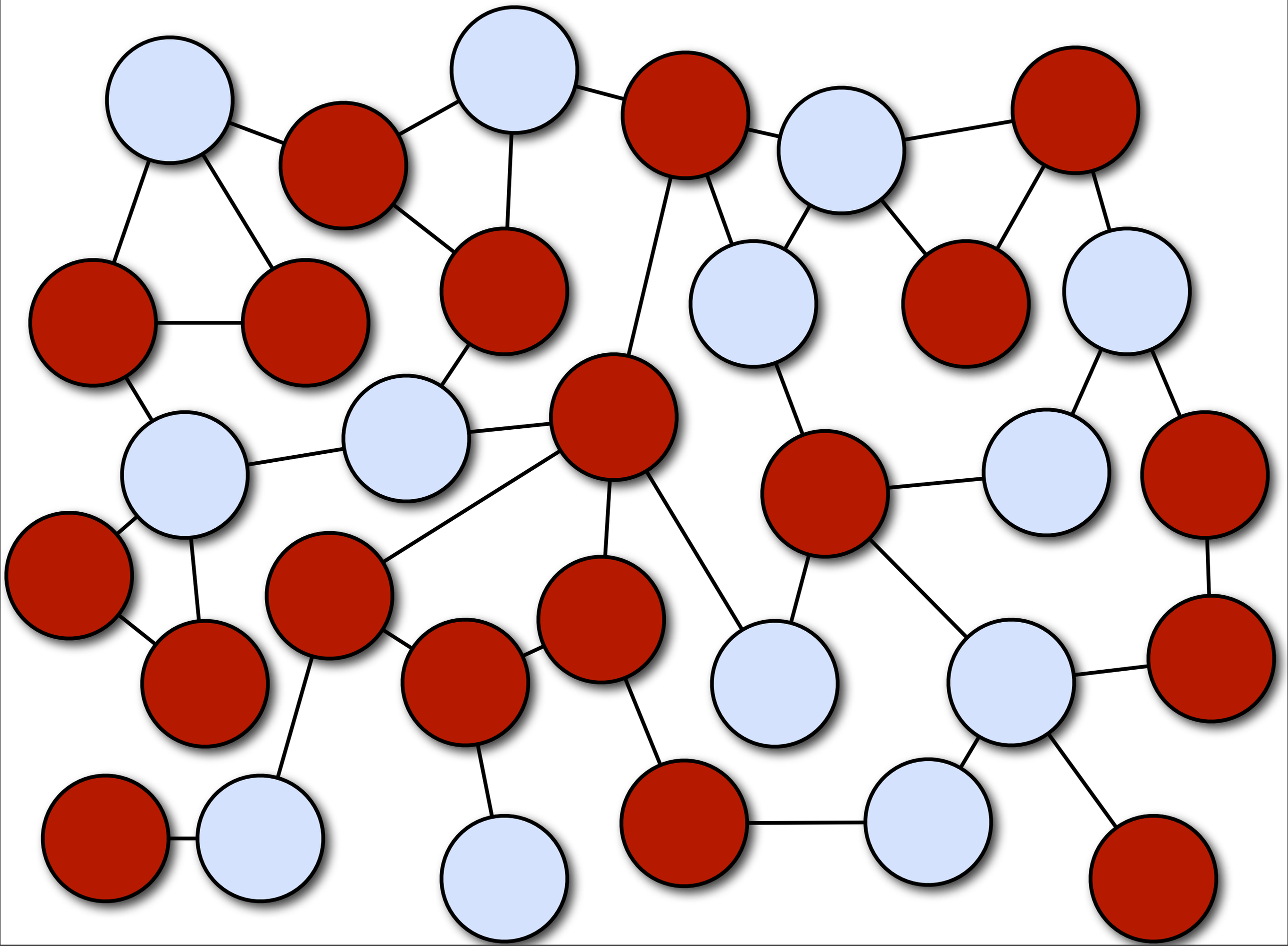


Which components have

critically important state

and

explicit error handling?



Classification of State

- Scratch data
- Static data
  - Supplied at boot time
  - Supplied by other components
- Dynamic data
  - Data possible to recompute
  - Input from other sources; data that is impossible to recompute

Classification of State

- Scratch data
- Static data
  - Supplied at boot time
  - Supplied by other components
- Dynamic data
  - Data possible to recompute
  - Input from other sources; data that is impossible to recompute

Classification of State

- Scra
- Stati
  - Sup
  - Sup
- Dyna
  - Dat

nts

e

Must be

protected

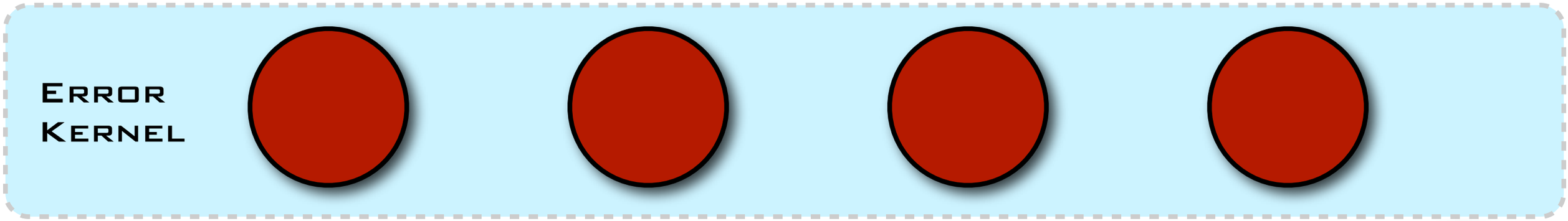
by any means

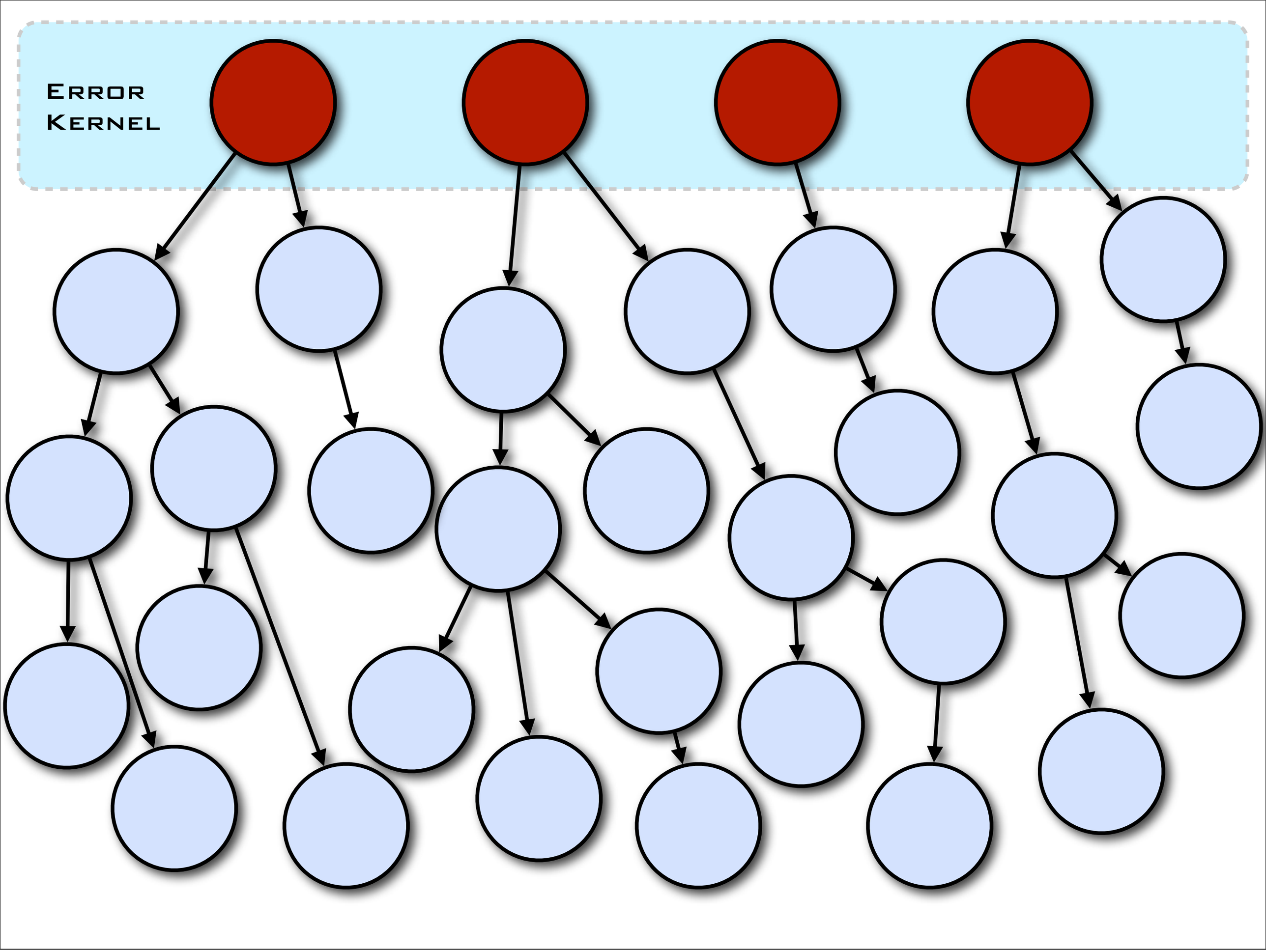
- Input *from other sources*; data that is *impossible to recompute*

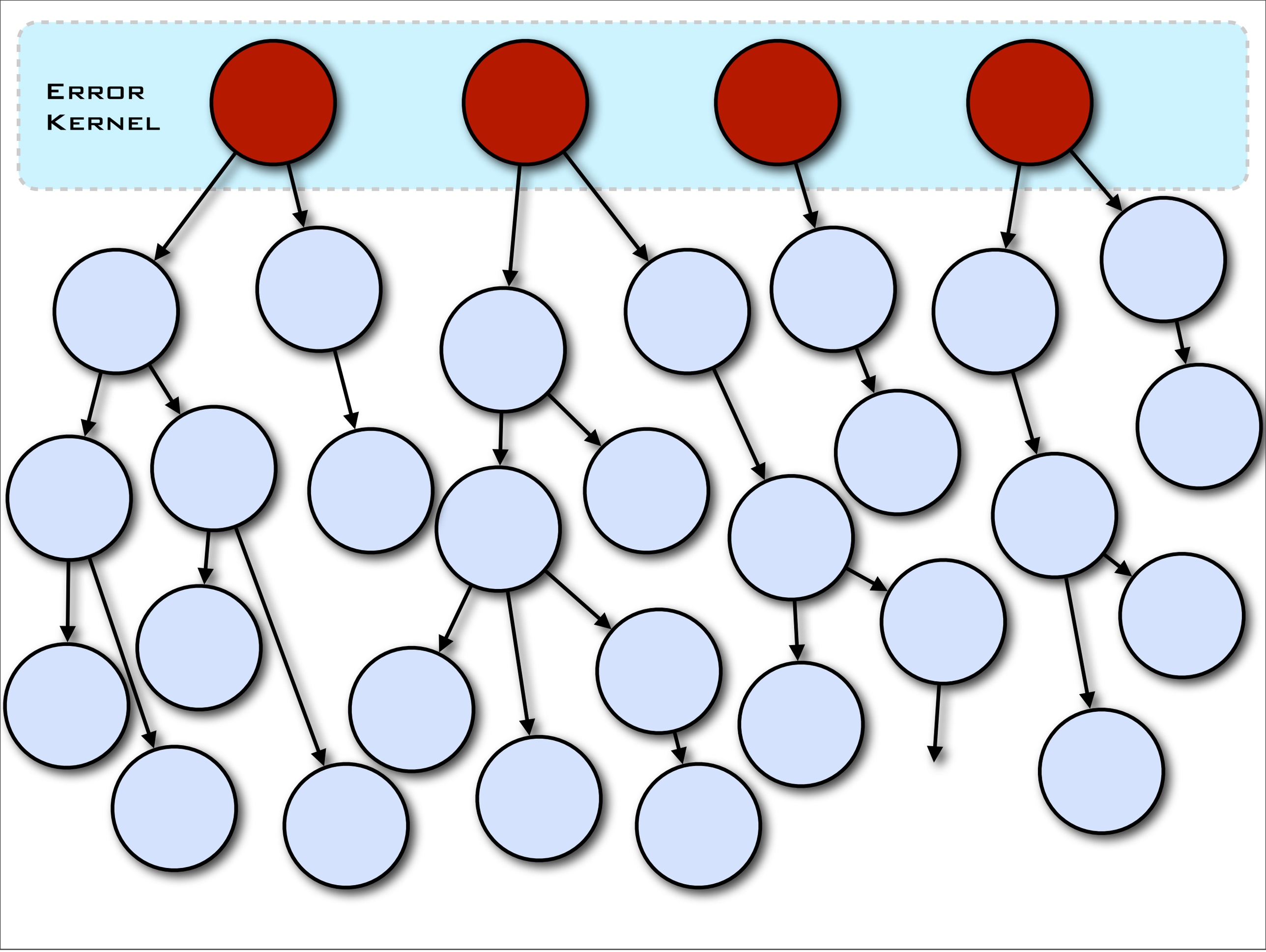
Fault-tolerant onion-layered Error Kernel

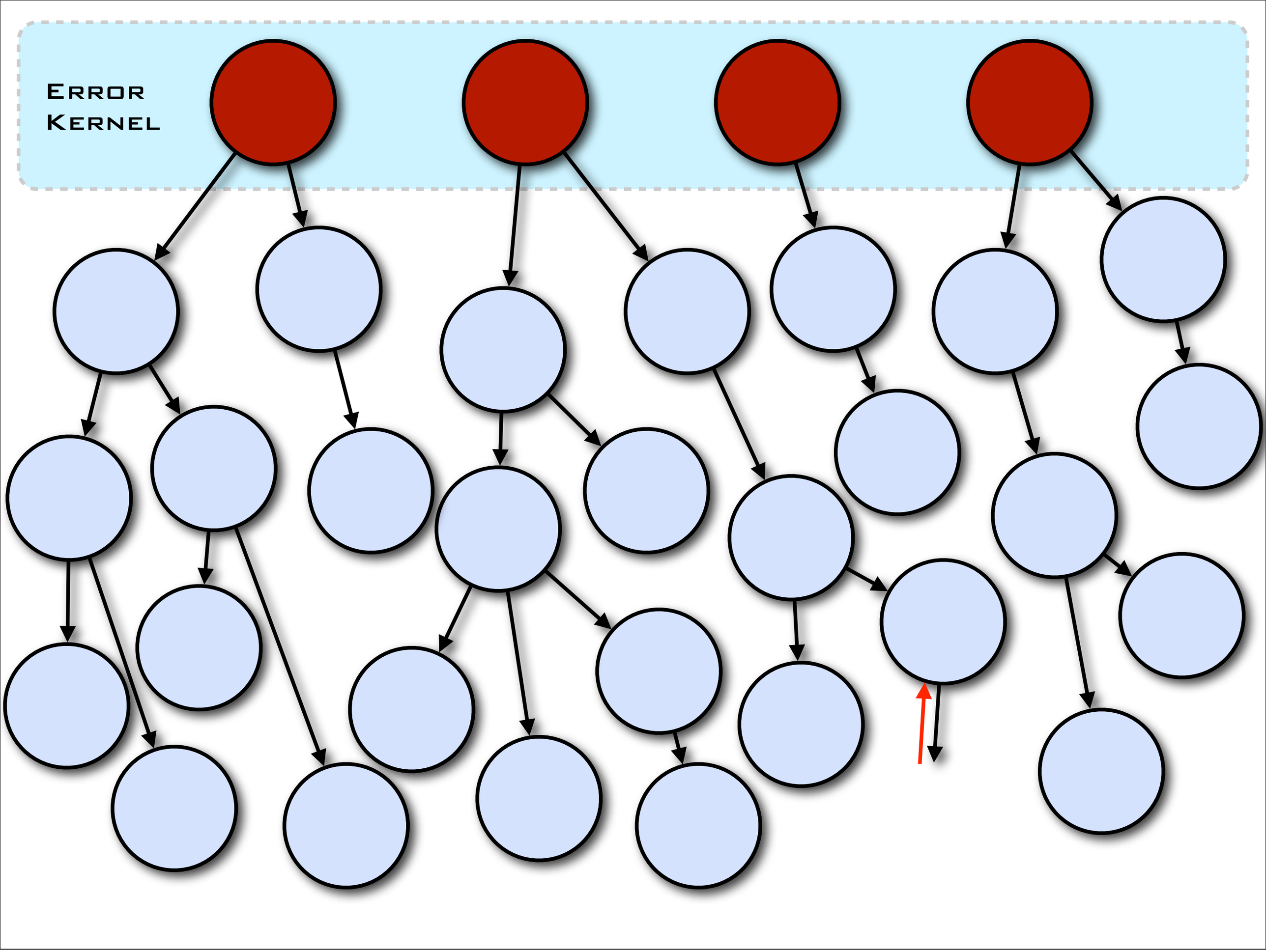
An actor-based application with explicit error handling and critically important state in an

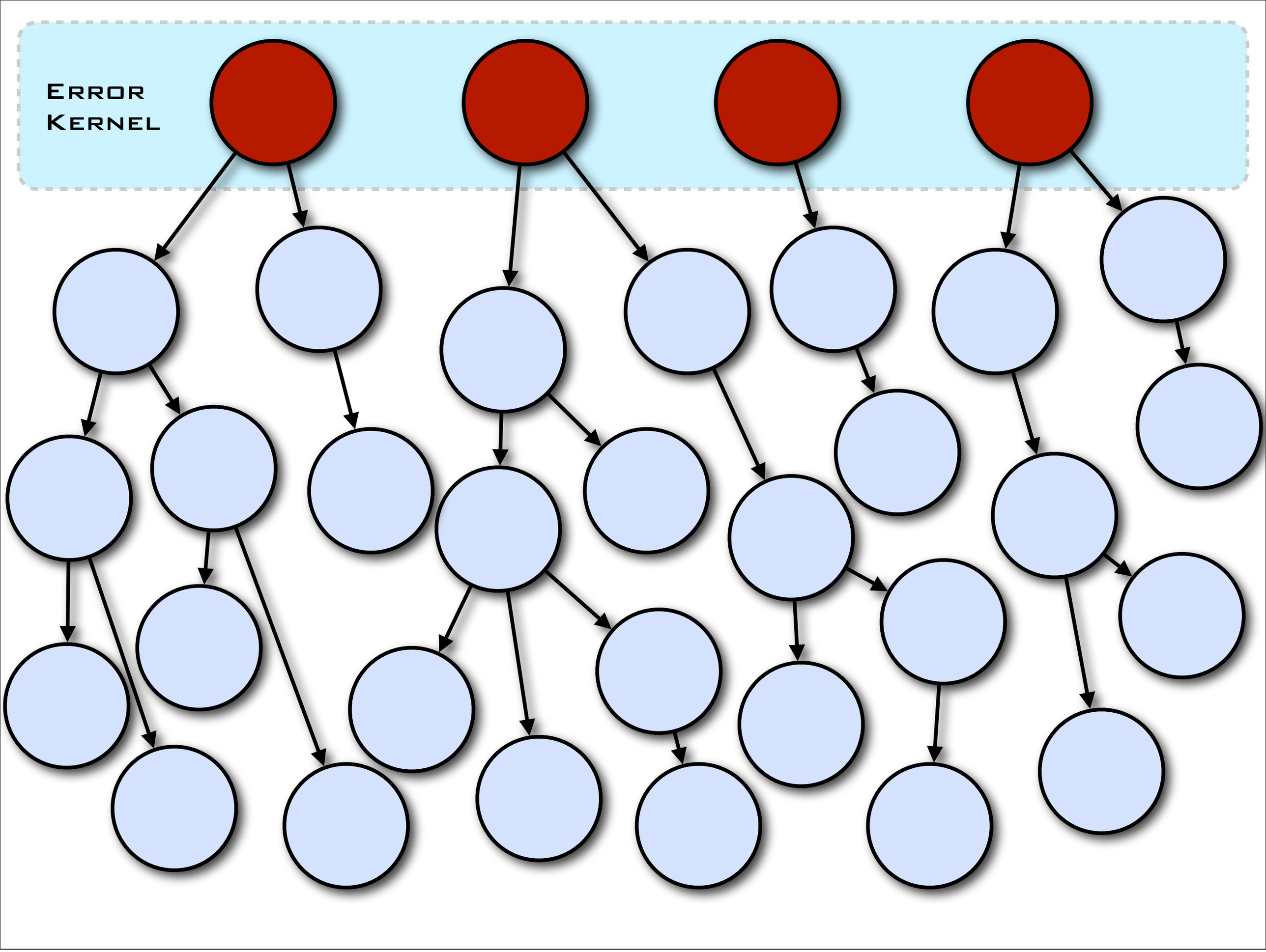
Error Kernel

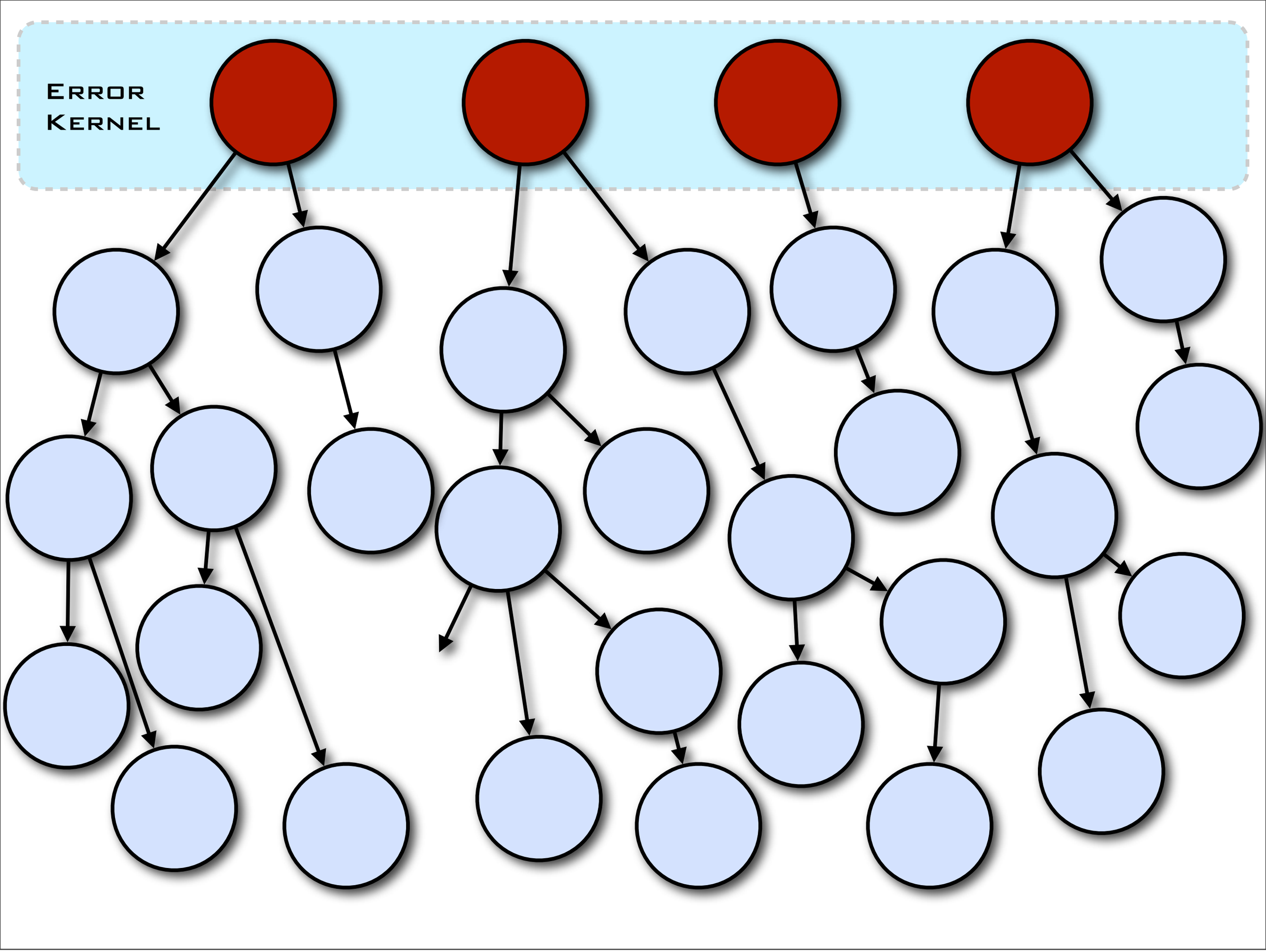

Error
Kernel

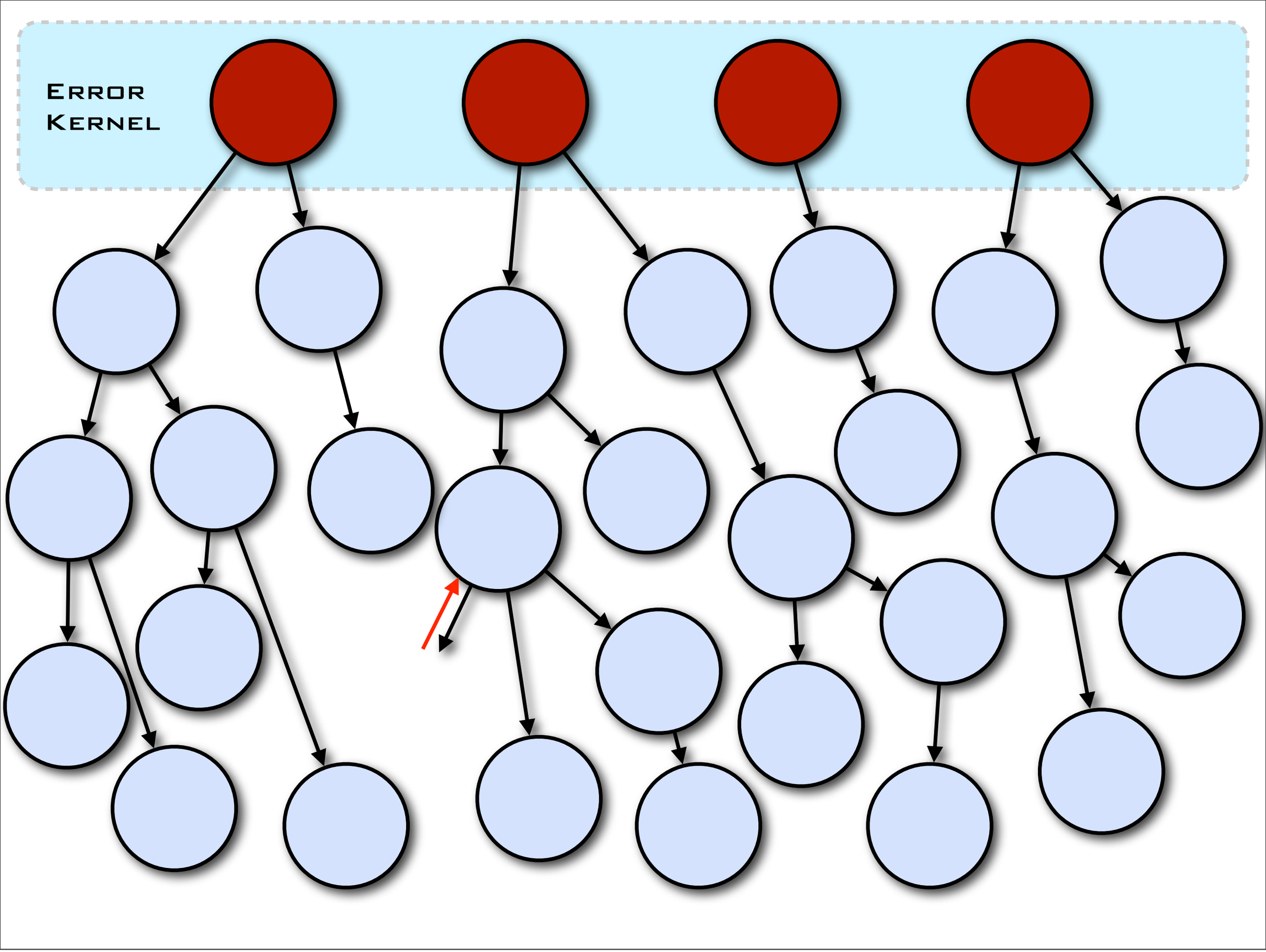

Error
Kernel

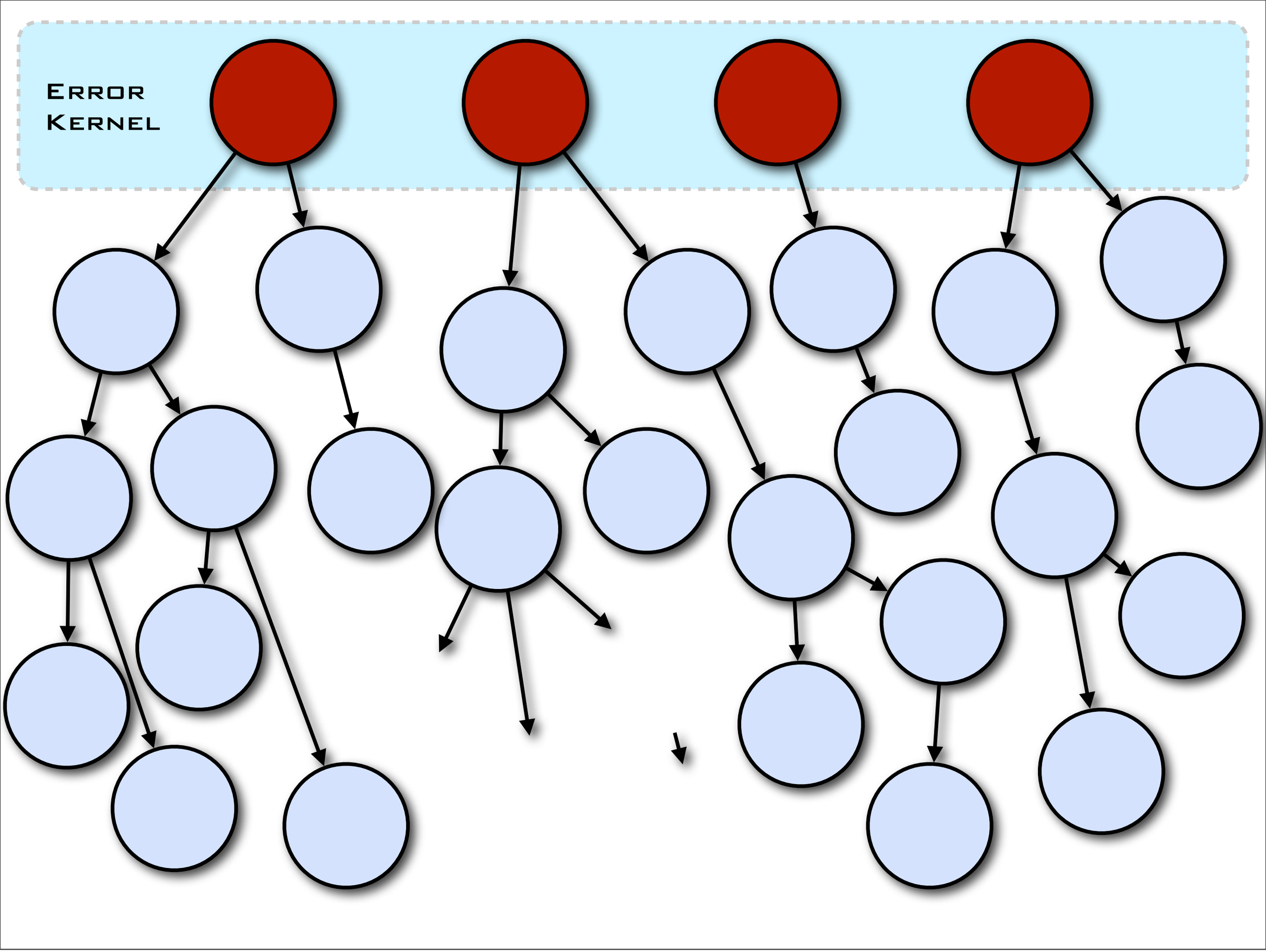

Error
Kernel

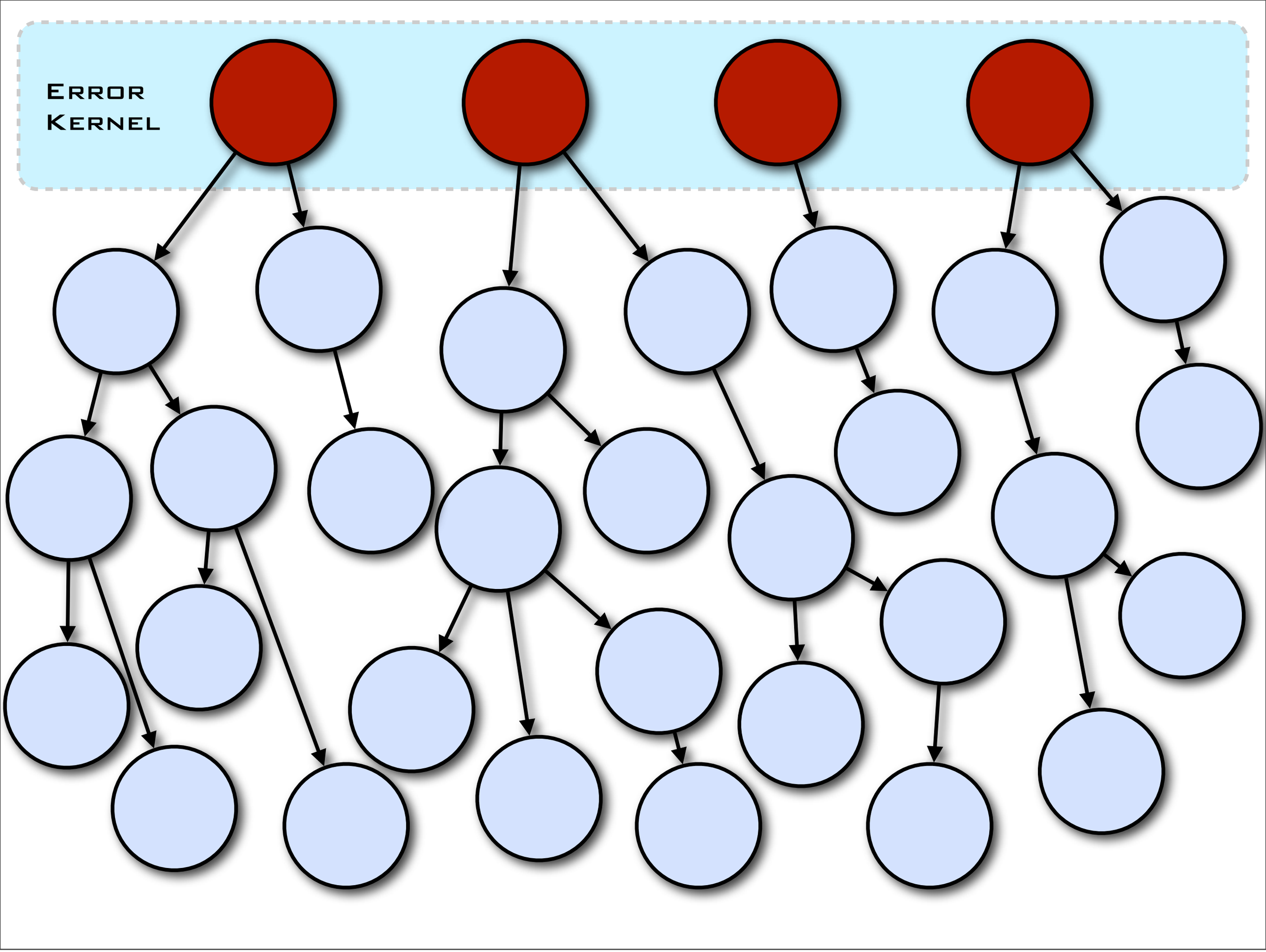

Error
Kernel

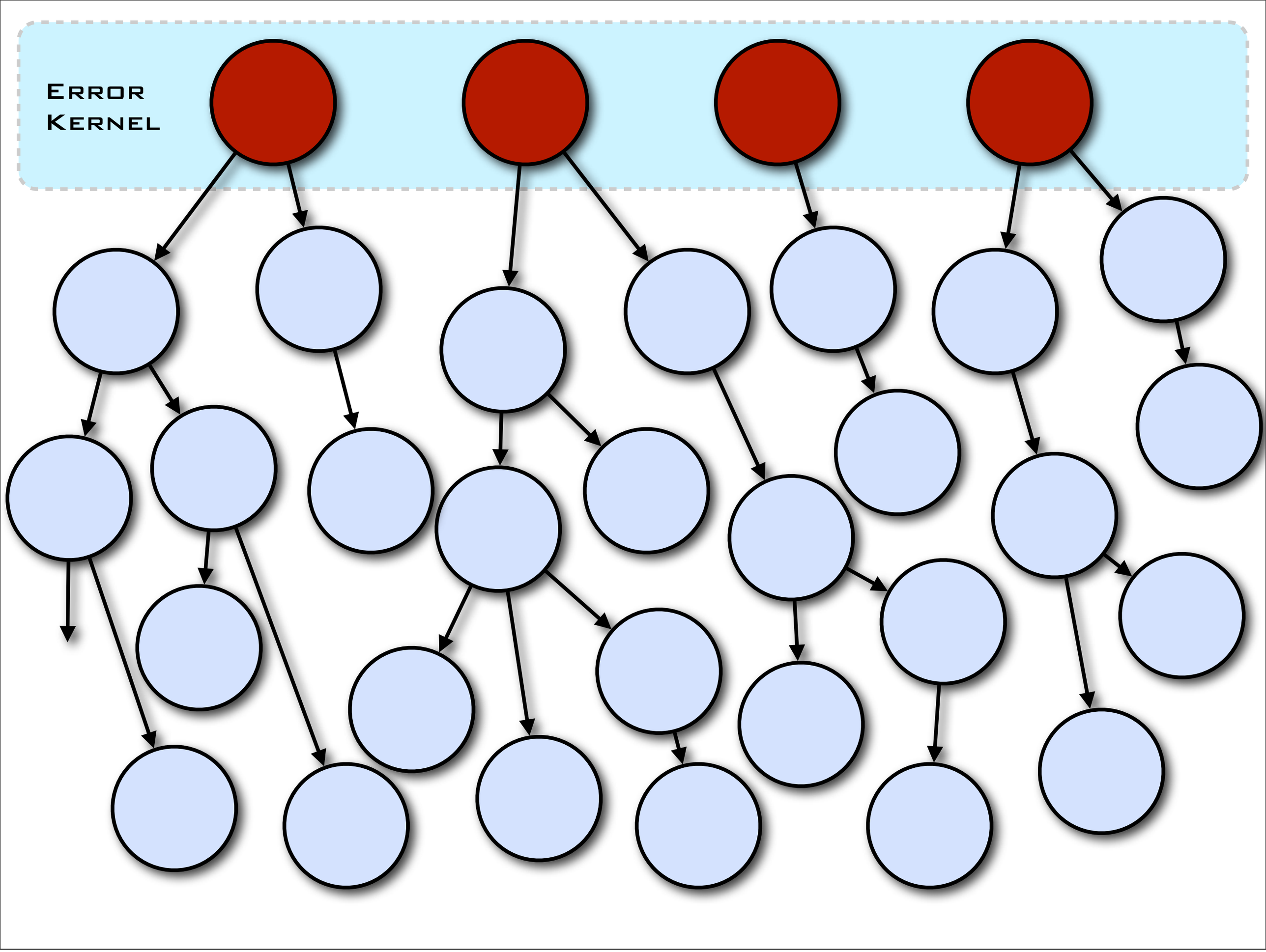

Error
Kernel

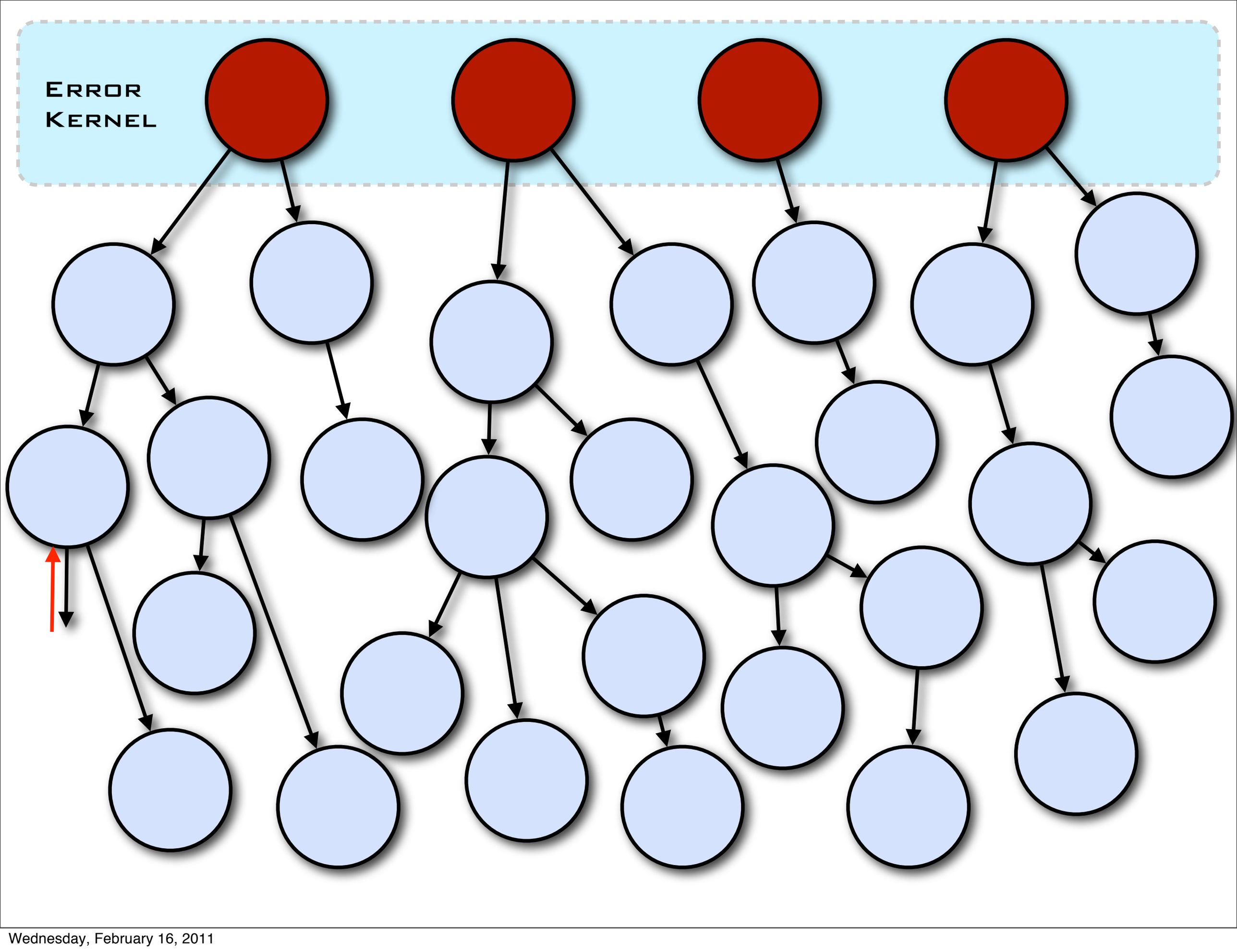

Error
Kernel

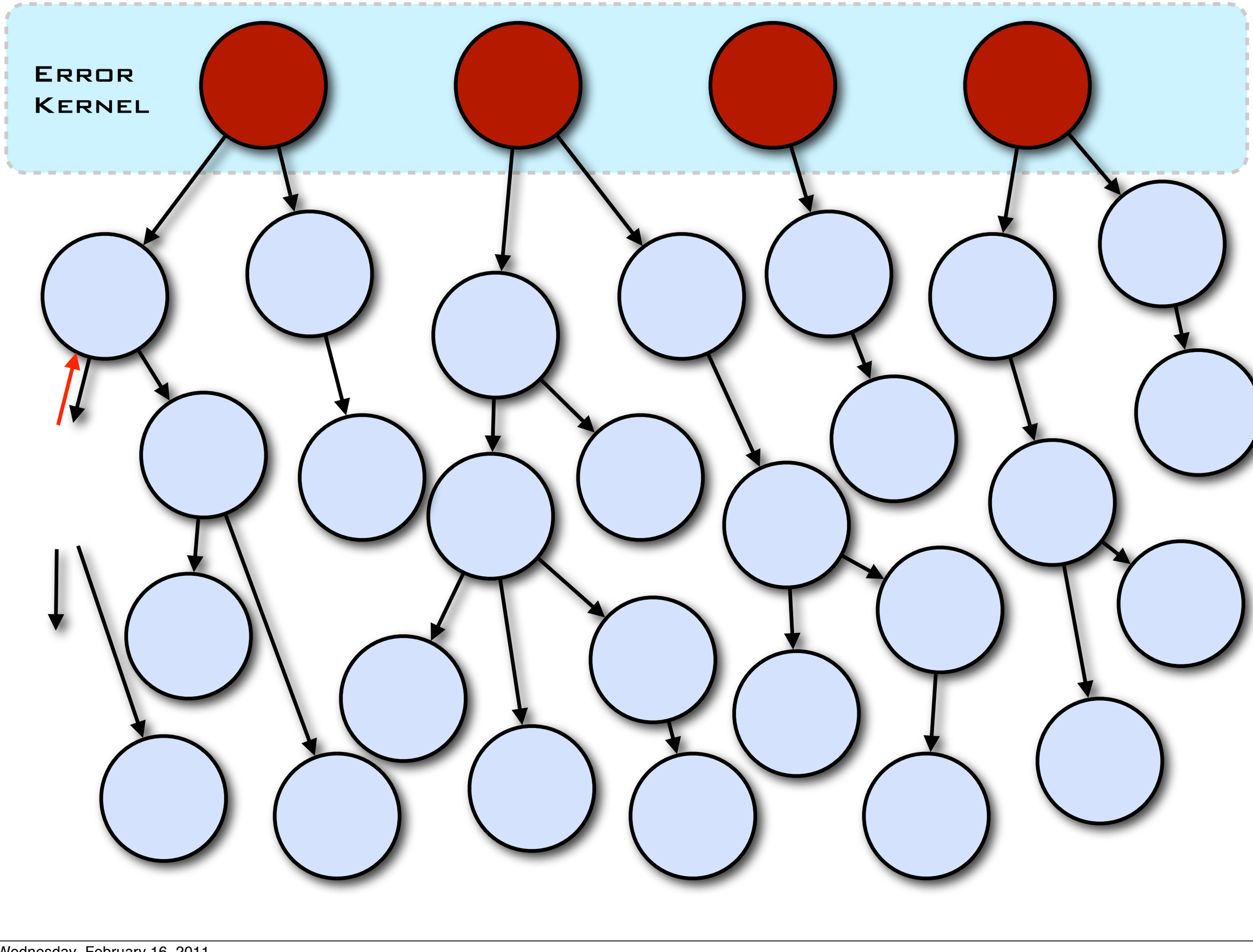

Error
Kernel

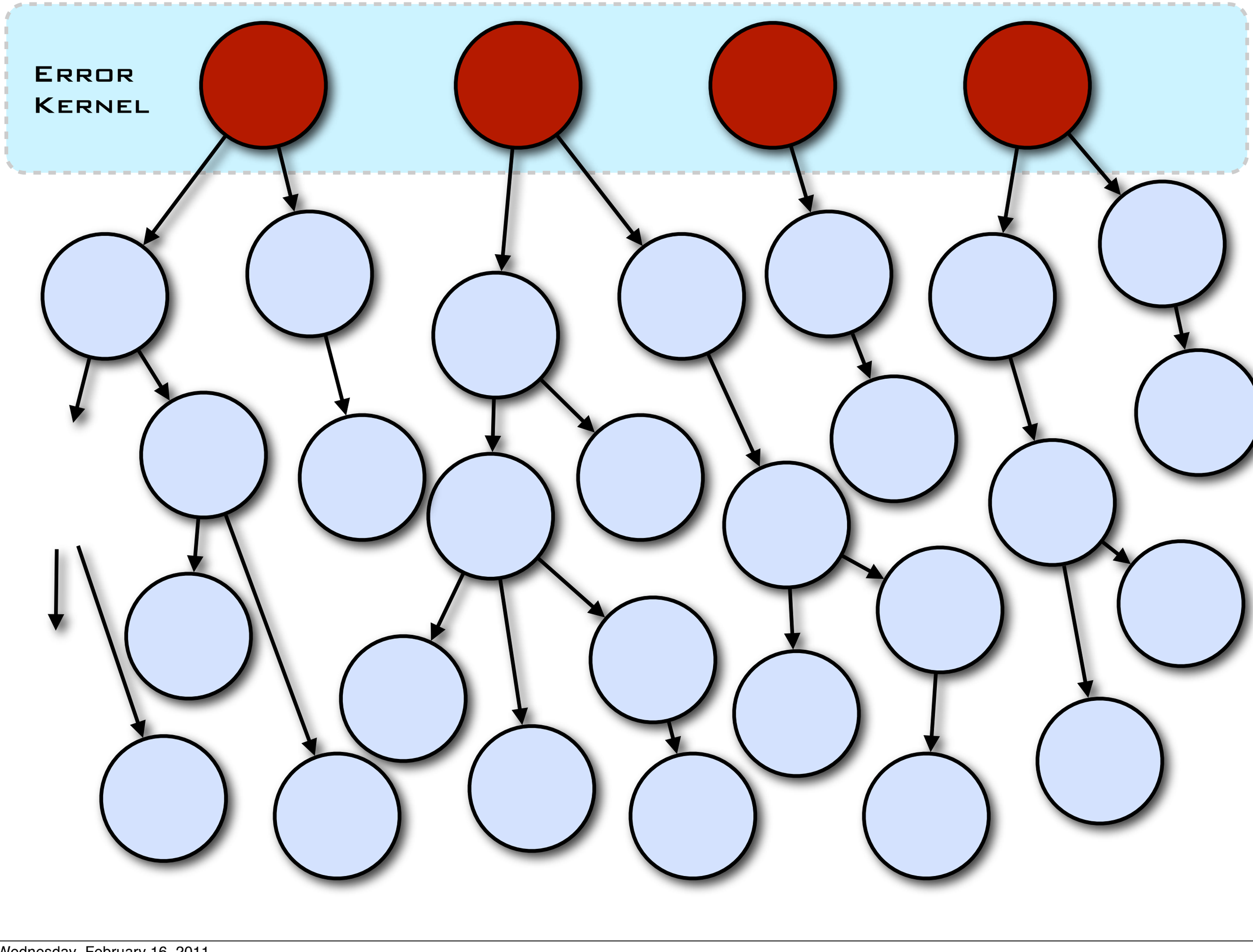

Error
Kernel

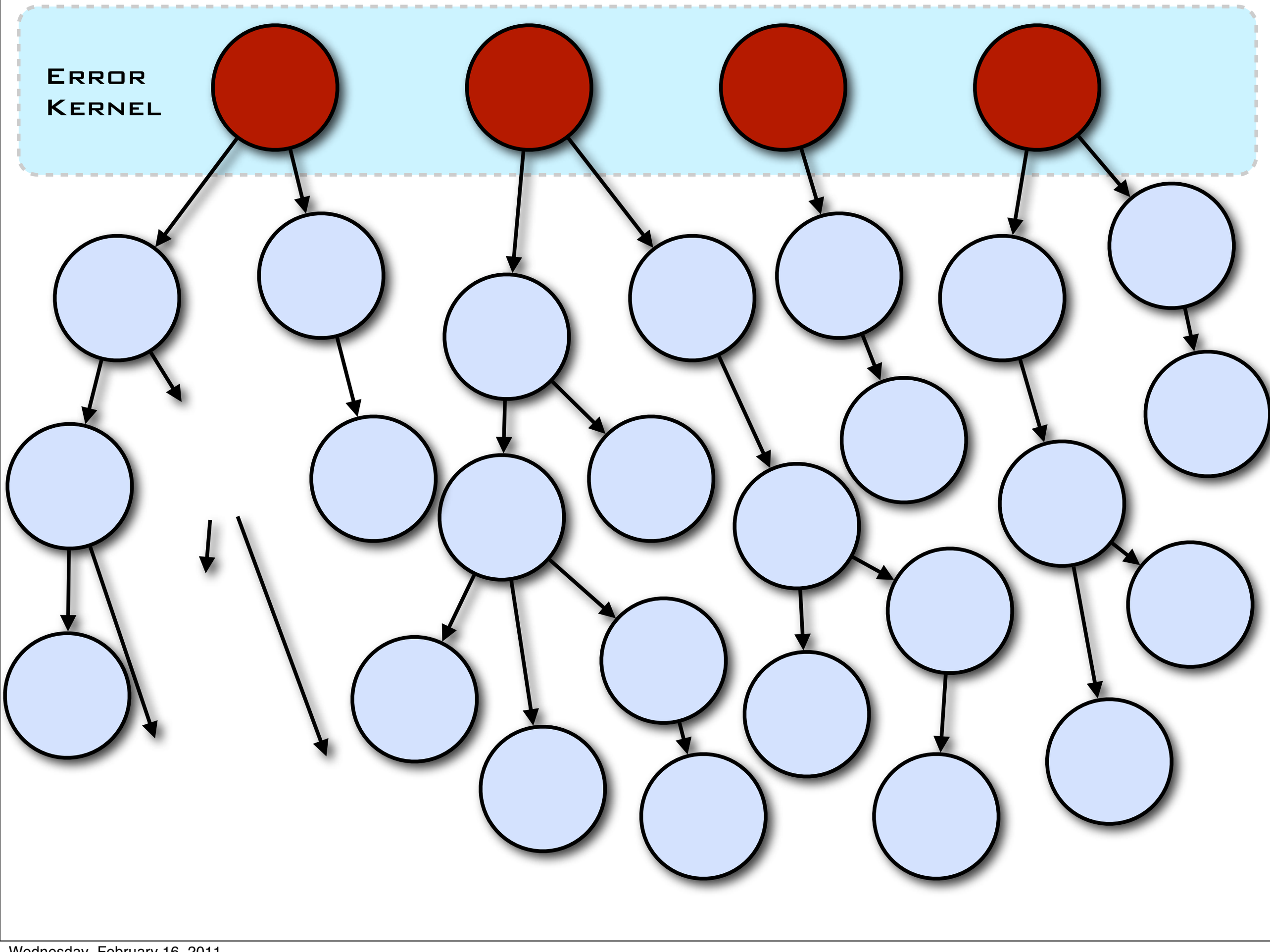

Error
Kernel


Error
Kernel

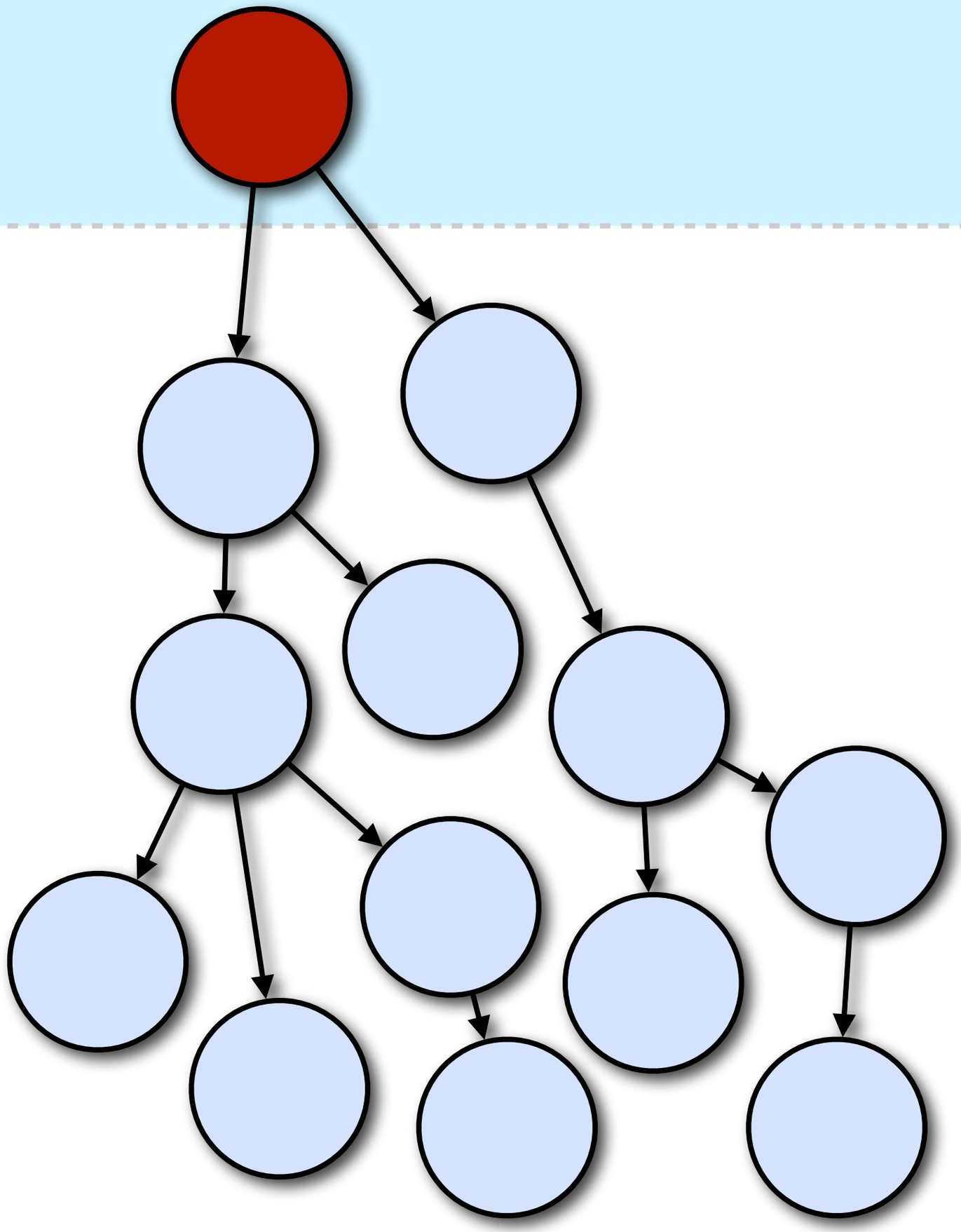

Error
Kernel

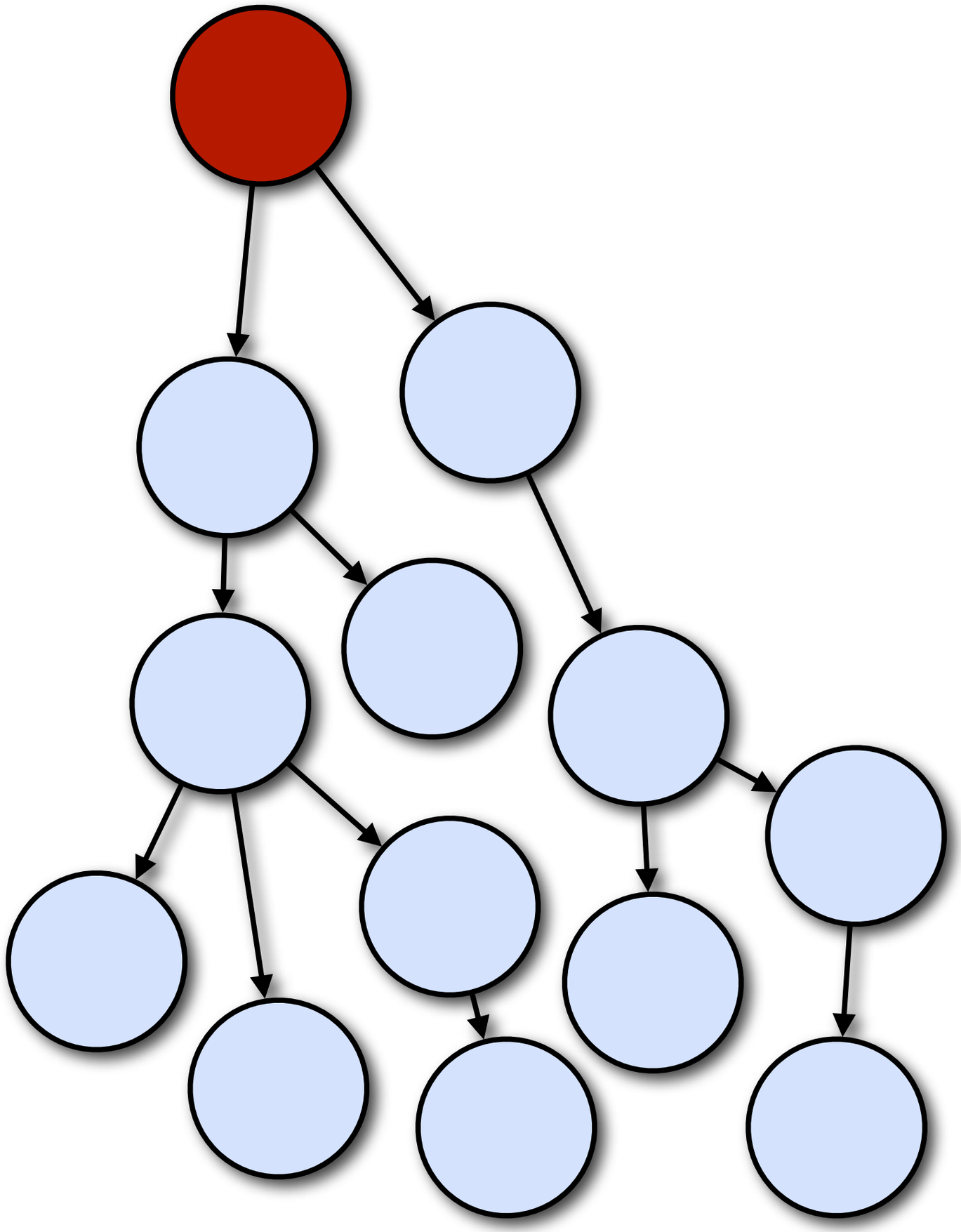

Error
Kernel

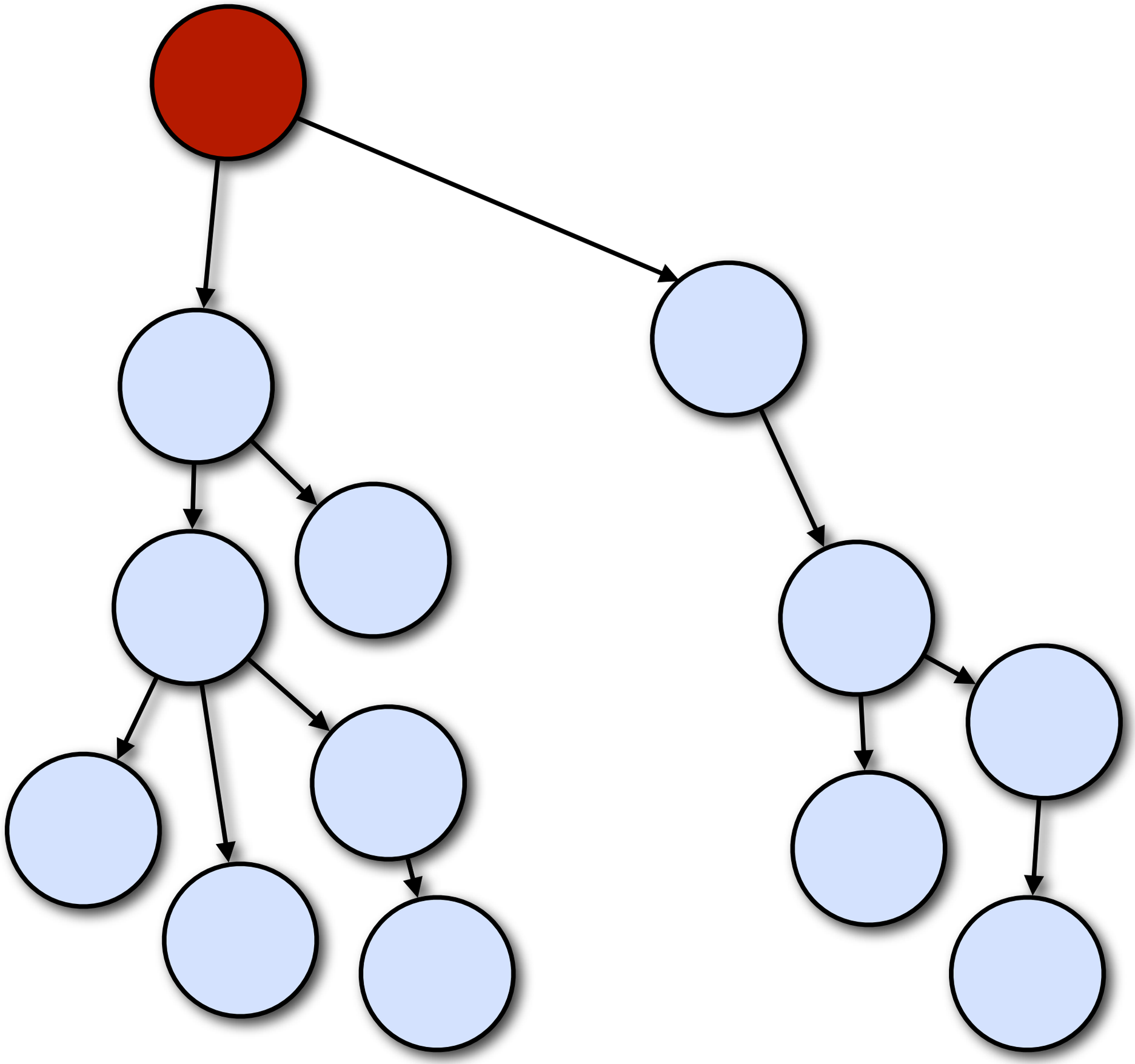

Error
Kernel

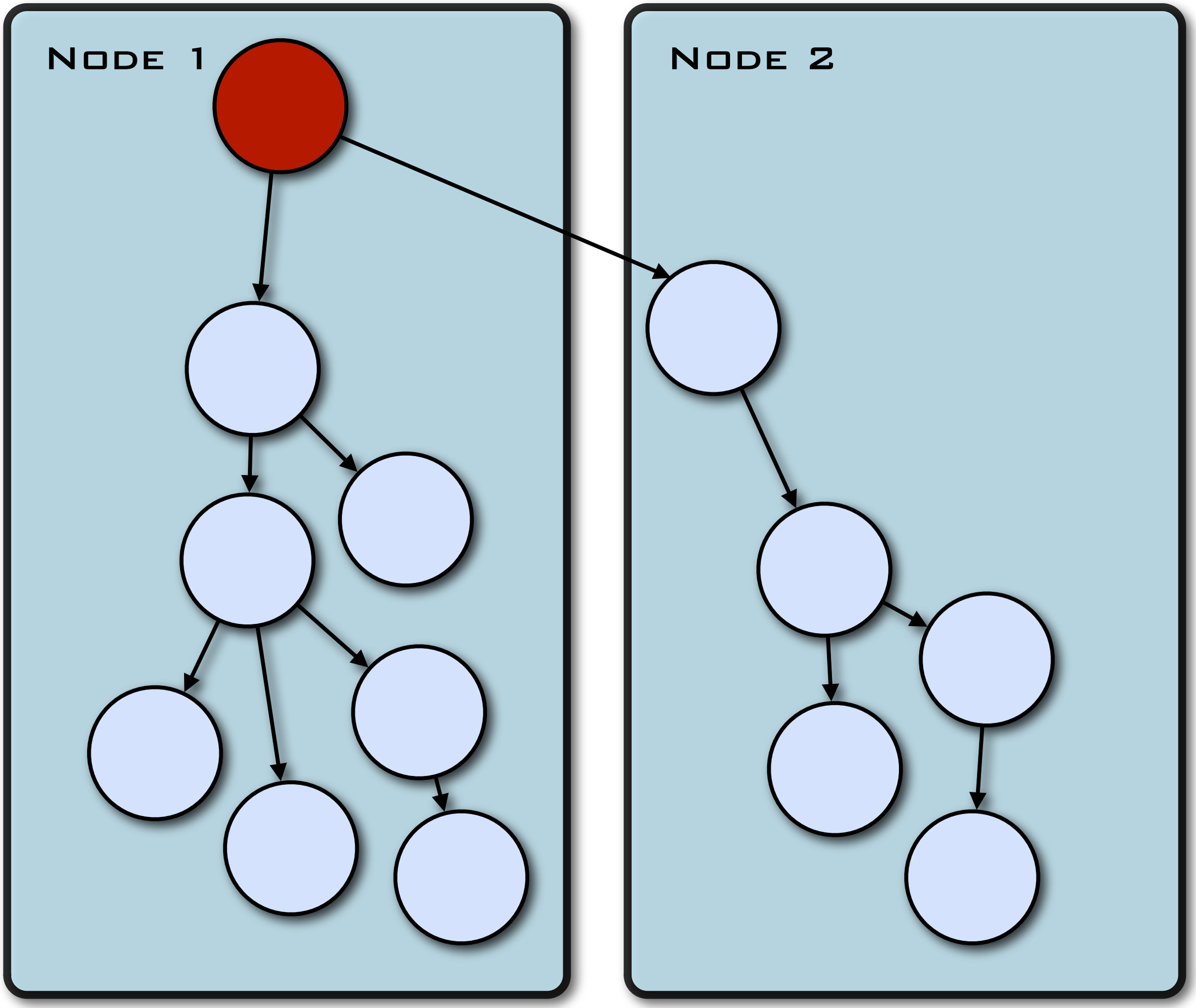

Error
Kernel

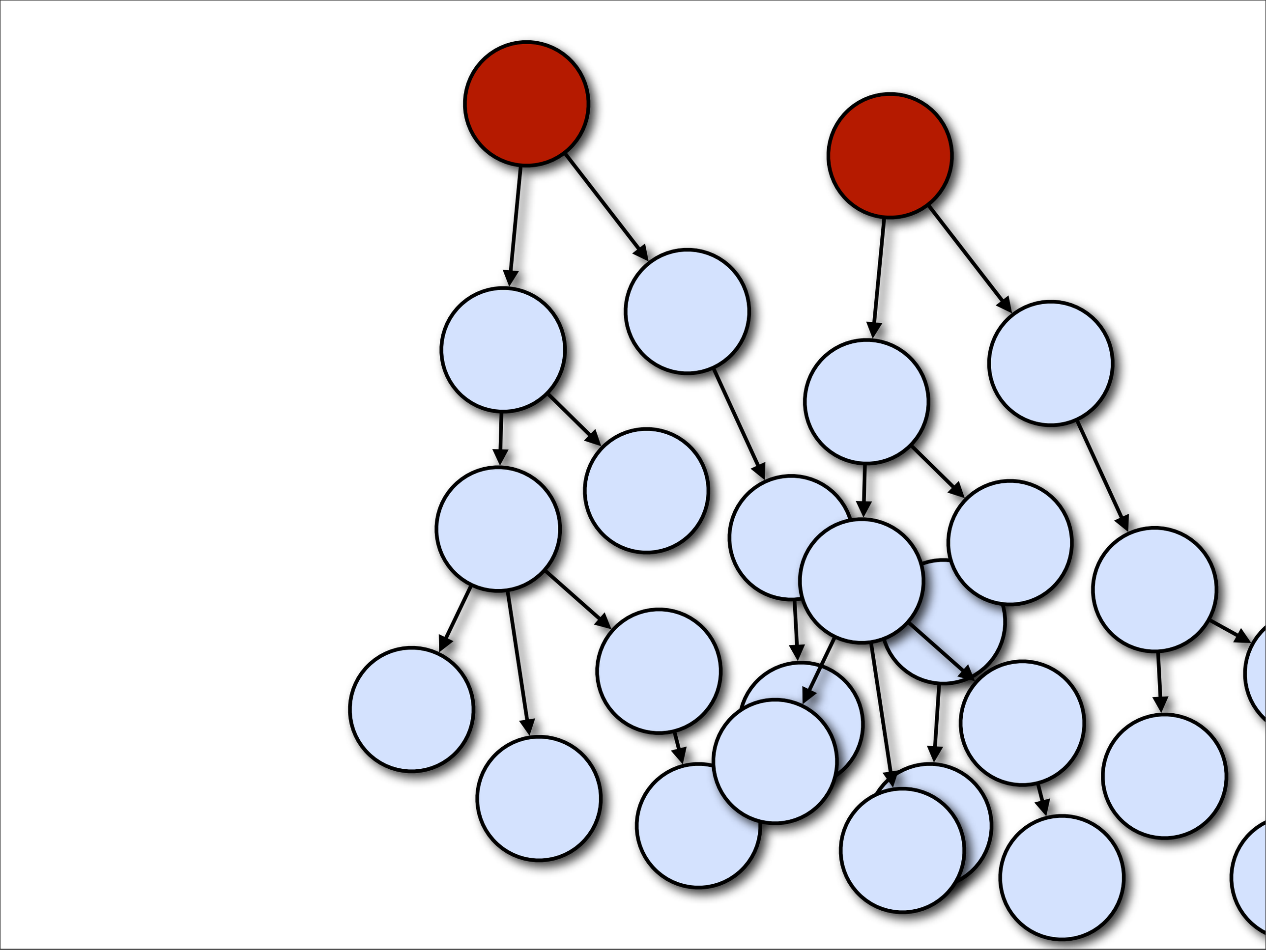
Error Kernel

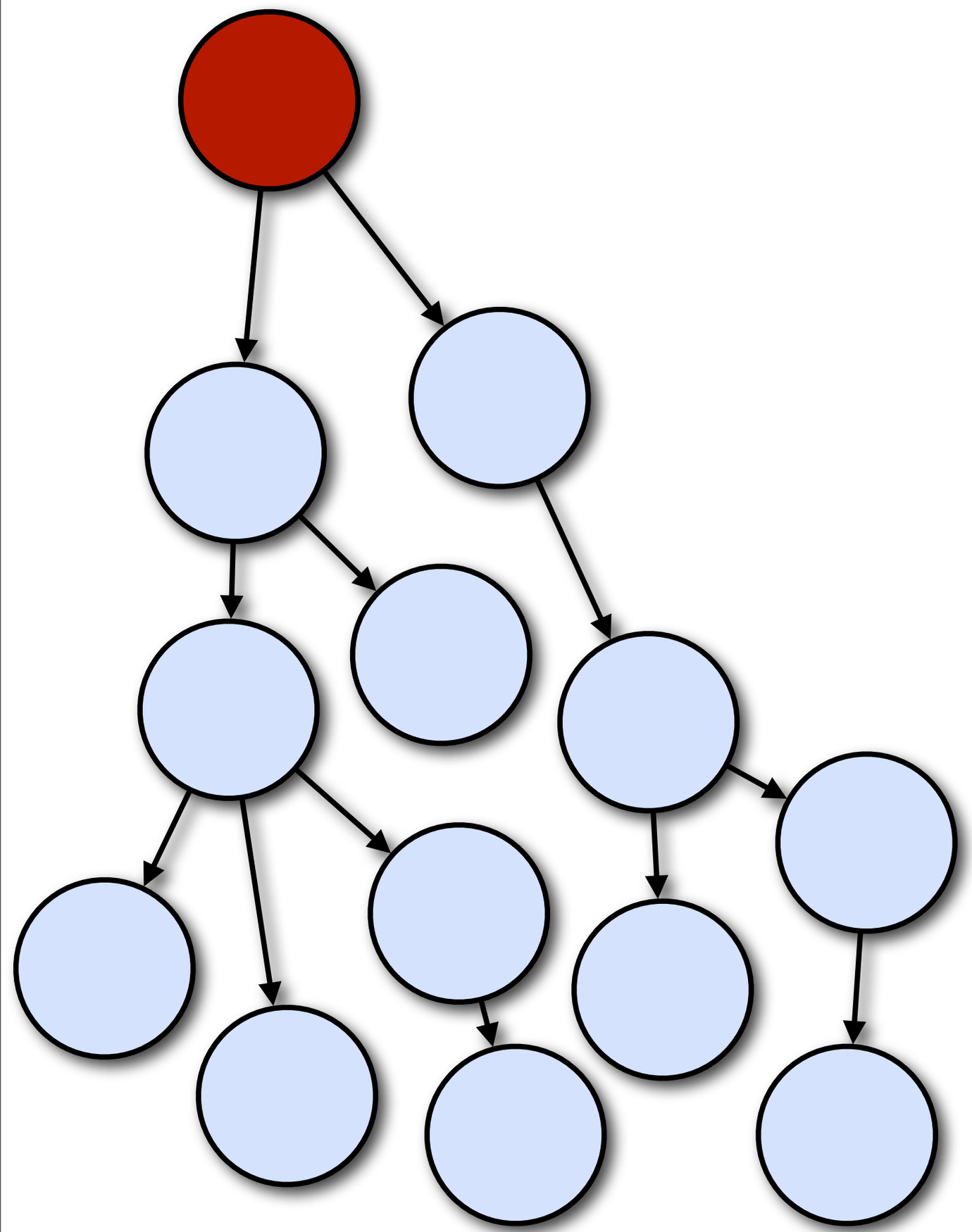


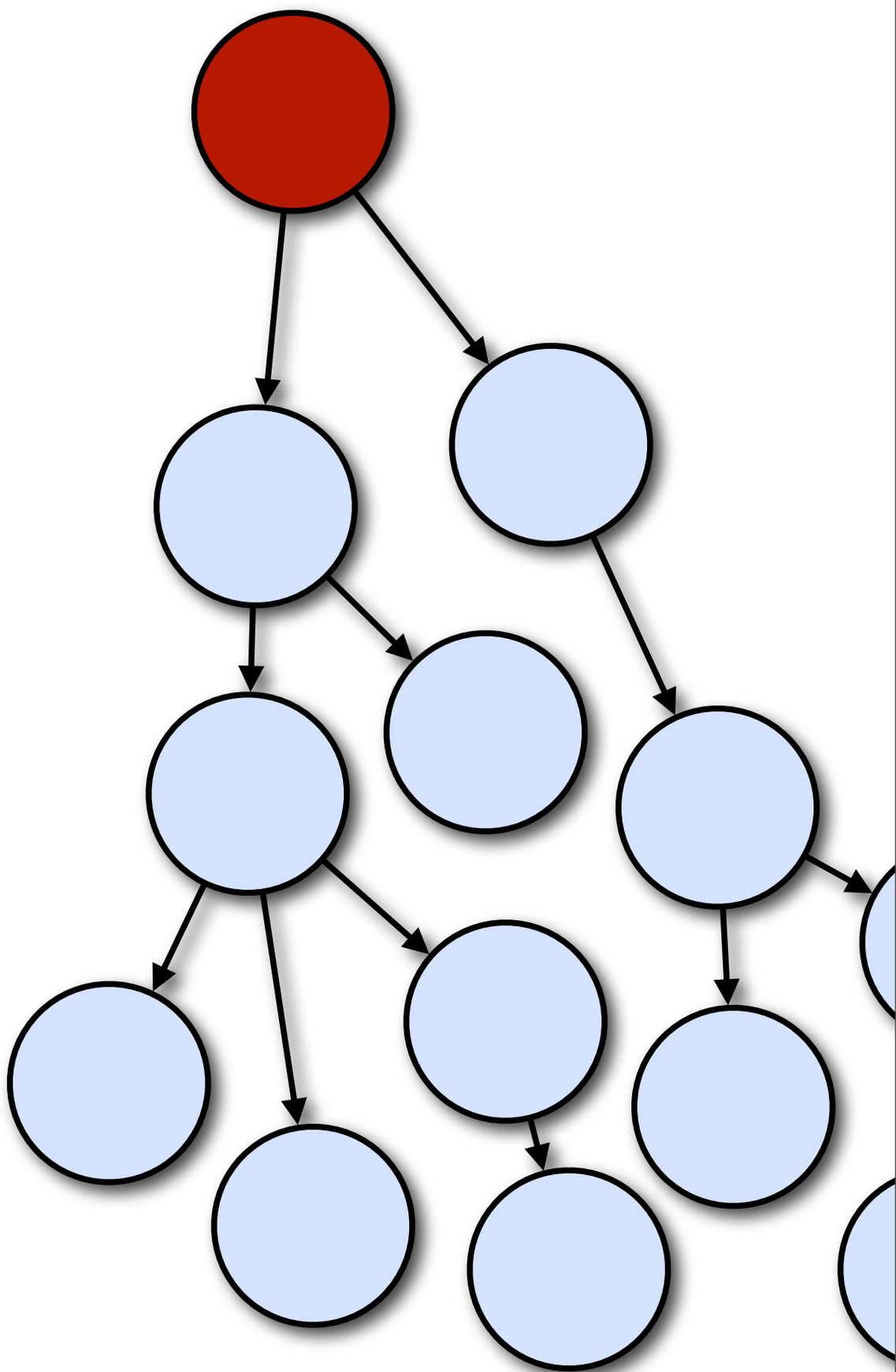


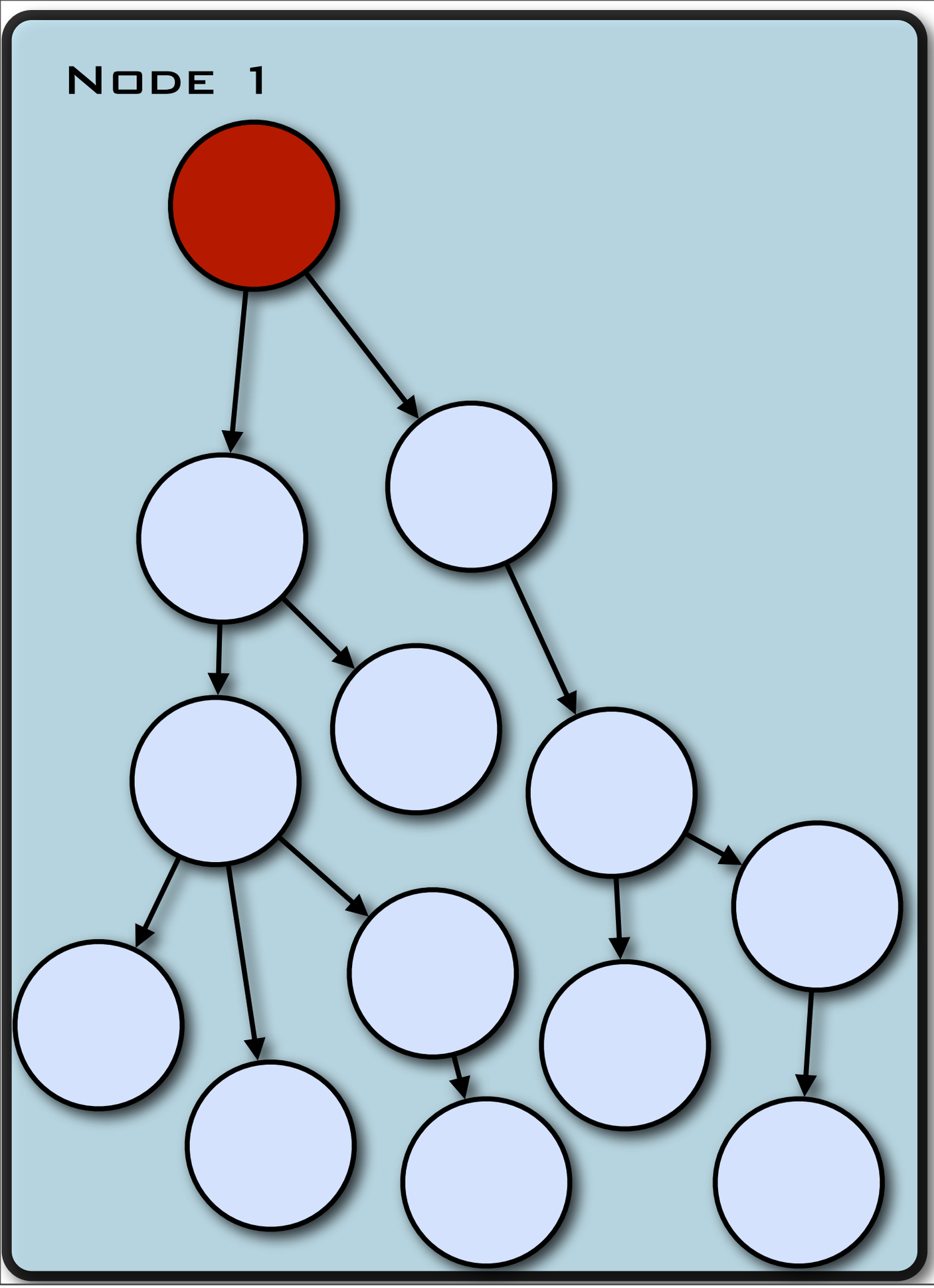


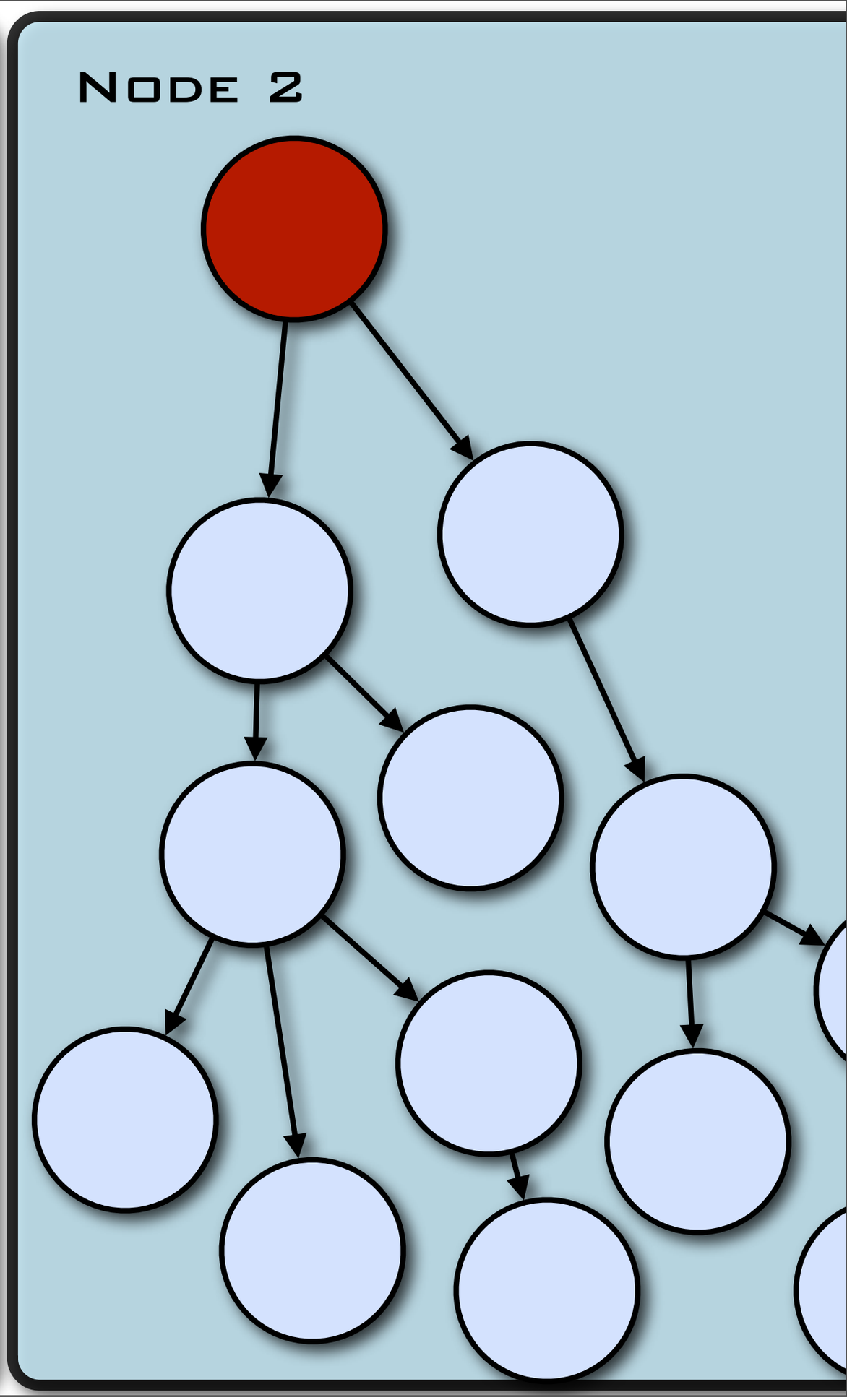

Node 1
Node 2

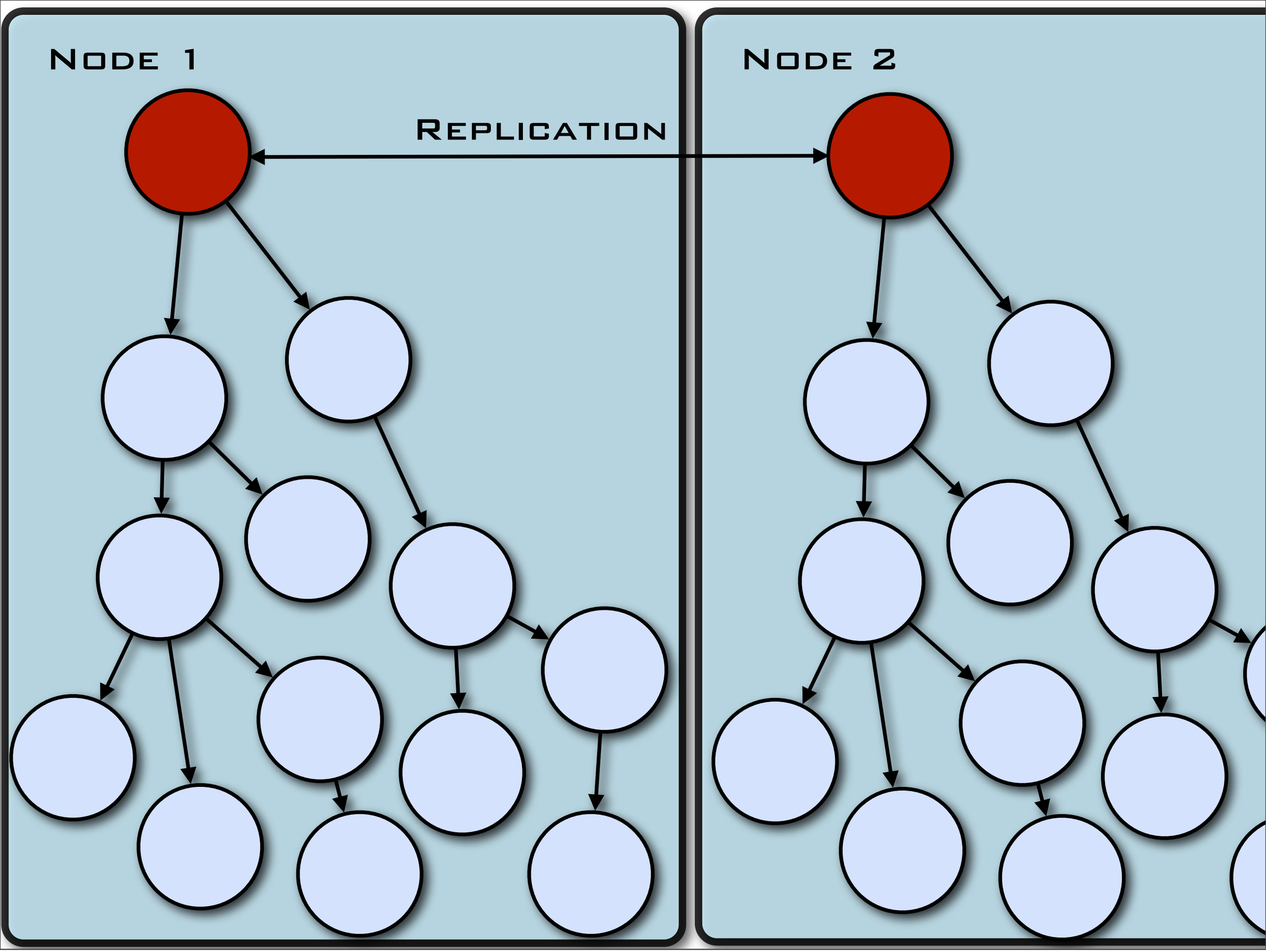







Node 1


Node 2


Node 1
Replication
Node 2

Fault handlers

```
AllForOneStrategy(
   problems,
   maxNrOfRetries,
   withinTimeRange)

OneForOneStrategy(
   problems,
   maxNrOfRetries,
   withinTimeRange)
```

Linking

```
link(actor)
unlink(actor)

startLink(actor)
spawnLink[MyActor]
```

Supervision

```
class Supervisor extends Actor {
  faultHandler = OneForOneStrategy(
    List(classOf[Throwable])
    5, 5000))

  def receive = {
    case Register(actor) =>
      link(actor)
  }
}
```

Manage failure

```
class FaultTolerantService extends Actor {
  ...
  override def preRestart(reason: Throwable) = {
    ... // clean up before restart
  }
  override def postRestart(reason: Throwable) = {
    ... // init after restart
  }
}
```

Declarative config

```
Supervisor supervisor = Supervisor(
  SupervisorConfig(
    AllForOneStrategy(List(classOf[Exception]), 3, 1000),
        Supervise(
          actor1,
          Permanent) ::
        Supervise(
          actor2,
          Temporary) ::
        Nil
    ))
```

Remote Actors

Remote Server

```
// use host & port in config
Actor.remote.start()

Actor.remote.start("localhost", 2552)
```

Scalable implementation based on
NIO (Netty) & Protobuf

Two types of

remote actors

Client initiated & managed

Server initiated & managed

Client-managed

supervision works across nodes

```
import Actor._

val service = remote.actorOf[MyActor](host, port)

service ! message
```

Server-managed

register and manage actor on server
client gets “dumb” proxy handle

```
import Actor._

remote.register(“service:id”, actorOf[MyService])
```

server part

Server-managed

```
val service = remote.actorFor(
  “service:id”,
  “darkstar”,
  9999)

service ! message
```

client part

Server-managed

```
import Actor._

remote.register(actorOf[MyService])
remote.registerByUuid(actorOf[MyService])
remote.registerPerSession(
  “service:id”, actorOf[MyService])

remote.unregister(“service:id”)
remote.unregister(actorRef)
```

server part

Remoting Security

```
akka {
  remote {
    secure-cookie = "050E0A0D0D06010A00000900040D060F0C09060B"
    server {
      require-cookie = on
      untrusted-mode = on
    }
  }
}
```

Erlang-style secure cookie

Clustered Actors

Cloudy Akka

- Subscription-based cluster membership service
- Highly available cluster registry for actors
- Highly available centralized configuration service
- Automatic replication with automatic fail-over upon node crash
- Transparent and user-configurable load-balancing
- Transparent adaptive cluster rebalancing
- Leader election

Akka
Cluster Node

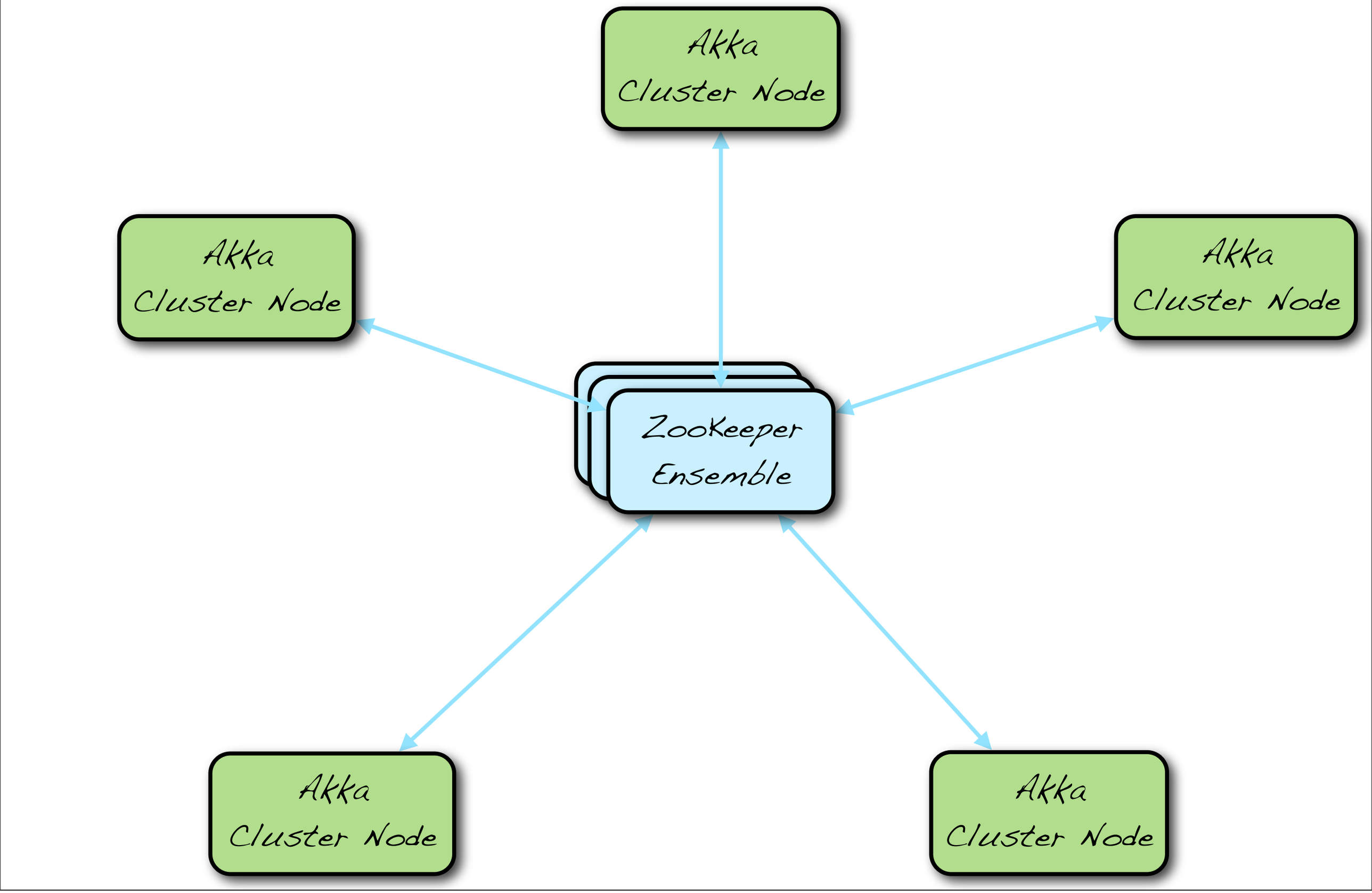
Akka
Cluster Node

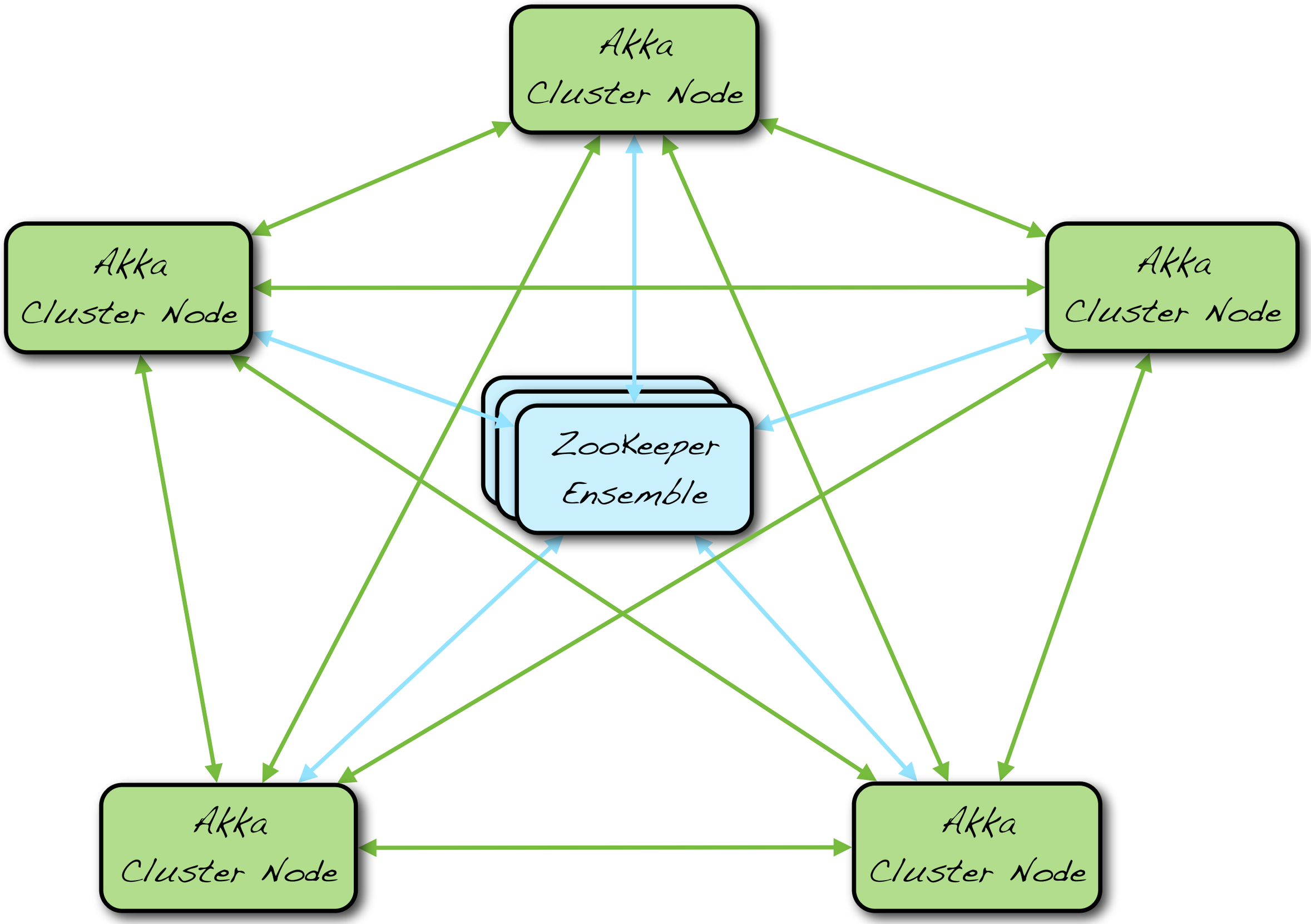
Akka
Cluster Node

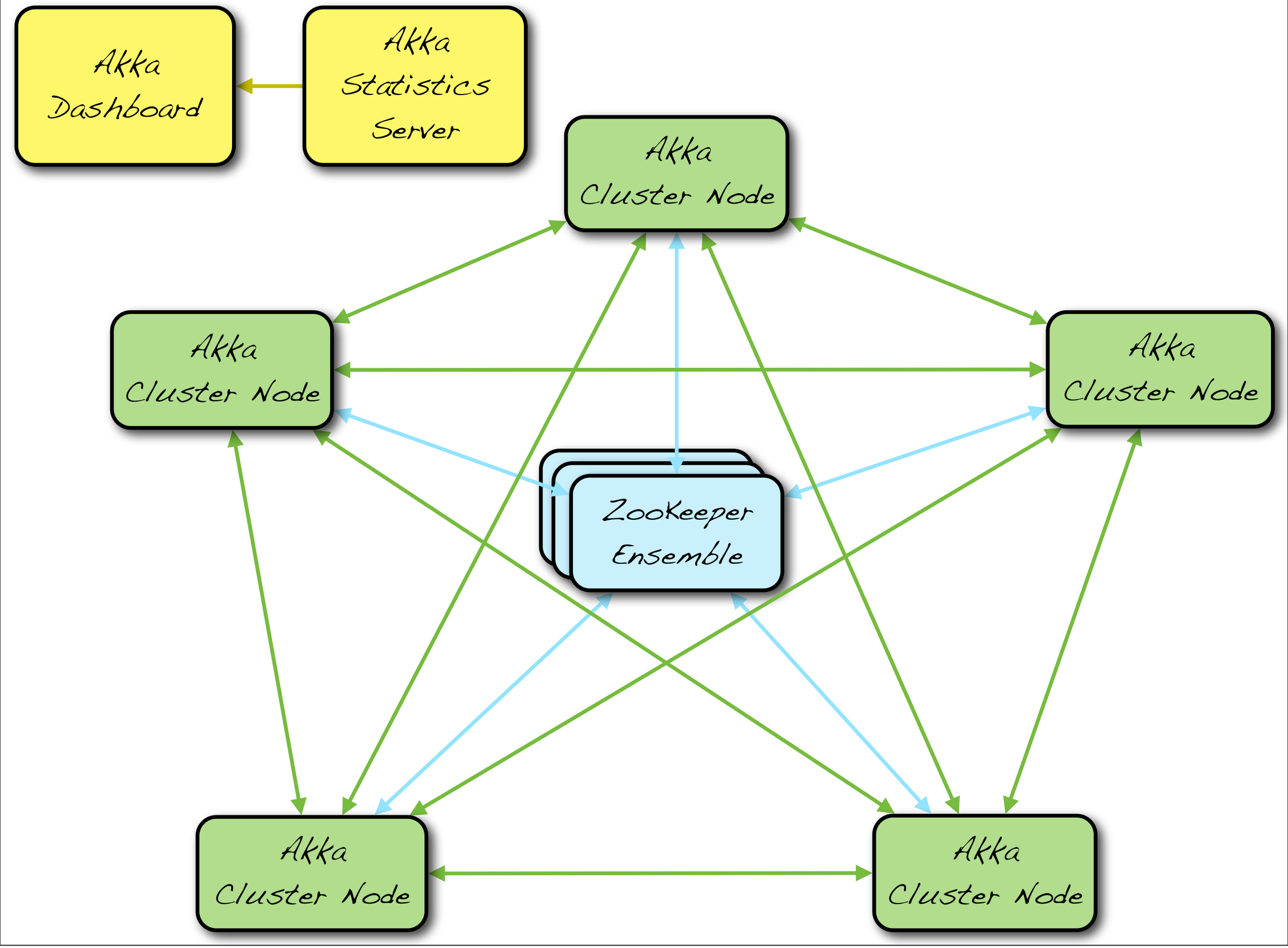
ZooKeeper
Ensemble

Akka
Cluster Node

Akka
Cluster Node


Akka
Cluster Node
Akka
Cluster Node
Akka
Cluster Node
ZooKeeper
Ensemble
Akka
Cluster Node
Akka
Cluster Node


Akka
Cluster Node
Akka
Cluster Node
Akka
Cluster Node
ZooKeeper
Ensemble
Akka
Cluster Node
Akka
Cluster Node


Akka
Dashboard
Akka
Statistics
Server
Akka
Cluster Node
Akka
Cluster Node
Akka
Cluster Node
ZooKeeper
Ensemble
Akka
Cluster Node
Akka
Cluster Node

Akka Dashboard

Akka Statistics Server

Akka Cluster Node

Akka Cluster Node

Akka Cluster Node

ZooKeeper Ensemble

Akka Cluster Node

Akka Cluster Node

Example:
Clustered Actors

```
// Ping Pong messages
@serializable sealed trait PingPong
case object Ball extends PingPong
case object Stop extends PingPong
```

Example:
Clustered Actors

```
@serializable class PingActor extends Actor {
  var count = 0

  def receive = {
    case Ball =>
      if (count < NrOfPings) {
        println("----->> PING (%s)" format count)
        count += 1
        self reply Ball
      } else {
        self.sender.foreach(_ !! Stop)
        self.stop
      }
  }
}
```

Example:
Clustered Actors

```
@serializable class PongActor extends Actor {
  def receive = {
    case Ball =>
      self reply Ball
    case Stop =>
      self.stop
  }
}
```

Example:
Clustered Actors

```
Cluster.startLocalCluster()

val localNode   = Cluster.newNode(NodeAddress(CLUSTER_NAME, "node0", port = 9991)).start
val remoteNodes = Cluster.newNode(NodeAddress(CLUSTER_NAME, "node1", port = 9992)).start ::
                  Cluster.newNode(NodeAddress(CLUSTER_NAME, "node2", port = 9993)).start ::
                  Cluster.newNode(NodeAddress(CLUSTER_NAME, "node3", port = 9994)).start ::
                  Cluster.newNode(NodeAddress(CLUSTER_NAME, "node4", port = 9995)).start :: Nil
```

Example:
Clustered Actors

Example:
Clustered Actors

```
localNode store (classOf[PingActor])
```

Example:
Clustered Actors

```
localNode store (classOf[PingActor])
localNode store (classOf[PongActor], 5)
```

Example:
Clustered Actors

```
localNode store (classOf[PingActor])
localNode store (classOf[PongActor], 5)

val ping = localNode.use[PingActor](actorId = PING_SERVICE).head
```

Example:
Clustered Actors

```
localNode store (classOf[PingActor])
localNode store (classOf[PongActor], 5)

val ping = localNode.use[PingActor](actorId = PING_SERVICE).head
```

Example:
Clustered Actors

```
localNode store (classOf[PingActor])
localNode store (classOf[PongActor], 5)

val ping = localNode.use[PingActor](actorId = PING_SERVICE).head

val pong = localNode.ref(
```

Example:
Clustered Actors

```
localNode store (classOf[PingActor])
localNode store (classOf[PongActor], 5)

val ping = localNode.use[PingActor](actorId = PING_SERVICE).head

val pong = localNode.ref(
  actorId = PONG_SERVICE, router = Router.RoundRobin)
```

Example:
Clustered Actors

```
localNode store (classOf[PingActor])
localNode store (classOf[PongActor], 5)

val ping = localNode.use[PingActor](actorId = PING_SERVICE).head

val pong = localNode.ref(
  actorId = PONG_SERVICE, router = Router.RoundRobin)

implicit val replyTo = Some(pong) // set the reply address
```

Example:
Clustered Actors

```
localNode store (classOf[PingActor])
localNode store (classOf[PongActor], 5)

val ping = localNode.use[PingActor](actorId = PING_SERVICE).head

val pong = localNode.ref(
  actorId = PONG_SERVICE, router = Router.RoundRobin)

implicit val replyTo = Some(pong) // set the reply address
ping ! Ball                       // serve
```

How to run it?

- ☑ Deploy as dependency JAR in WEB-INF/lib etc.
- ☑ Run as stand-alone microkernel
- ☑ OSGi-enabled; drop in any OSGi container (Spring DM server, Karaf etc.)

...and much much more

HTTP

STM

FSM

Camel

JTA

Dataflow

Guice

OSGi

PubSub

AMQP

Spring

Persistence

Security

Use Akka from

Scala

Java

JRuby

Groovy

Akka

1.0

Released today

Get it and learn more

http://akka.io

EOF

Akka Camel

Camel: consumer

```
class MyConsumer extends Actor with Consumer {
  def endpointUri = "file:data/input"

  def receive = {
    case msg: Message =>
      log.info("received %s" format
      msg.bodyAs(classOf[String]))
  }
}
```

Camel: consumer

```
class MyConsumer extends Actor with Consumer {
  def endpointUri =
    "jetty:http://0.0.0.0:8877/camel/test"

  def receive = {
    case msg: Message =>
      reply("Hello %s" format
            msg.bodyAs(classOf[String]))
  }
}
```

Camel: producer

```
class CometProducer
  extends Actor with Producer {

  def endpointUri =
    "cometd://localhost:8111/test"
}
```

Camel: producer

```
val producer = actorOf[CometProducer].start

val time = "Current time: " + new Date
producer ! time
```