Above the Clouds:
Introducing Akka

Jonas Bonér
CEO @ Scalable Solutions
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

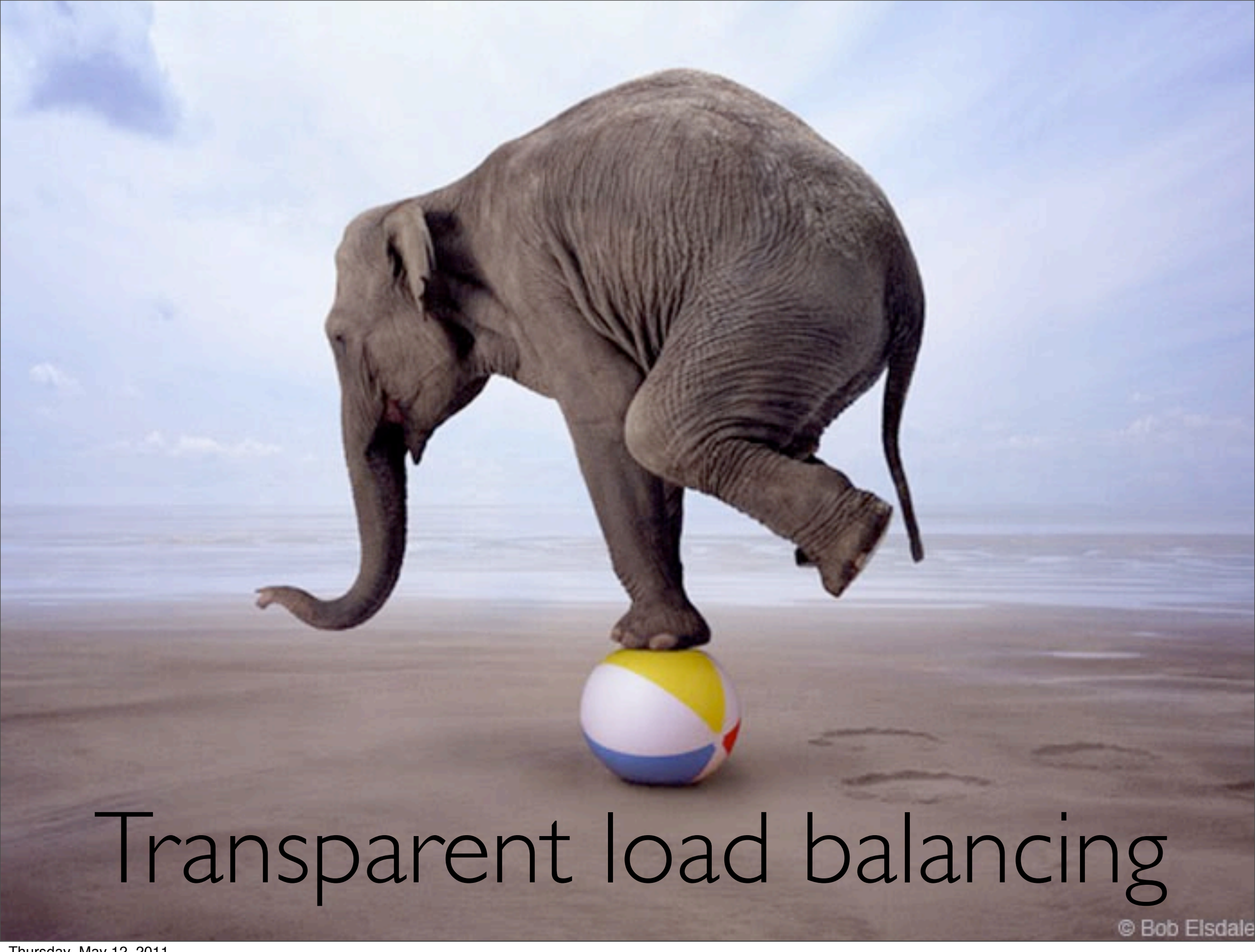
...with a single unified

- Programming model
- Runtime service

Manage system overload

Scale up & Scale out

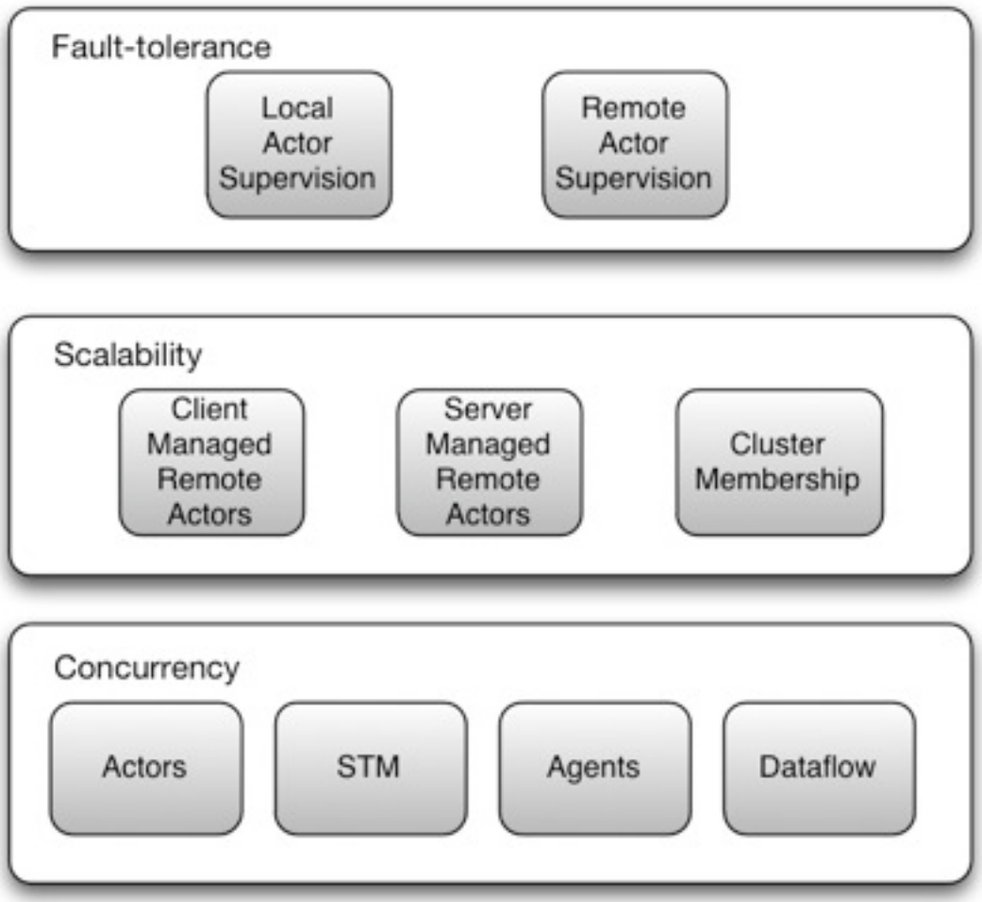
Replicate and distribute for fault-tolerance



Transparent load balancing

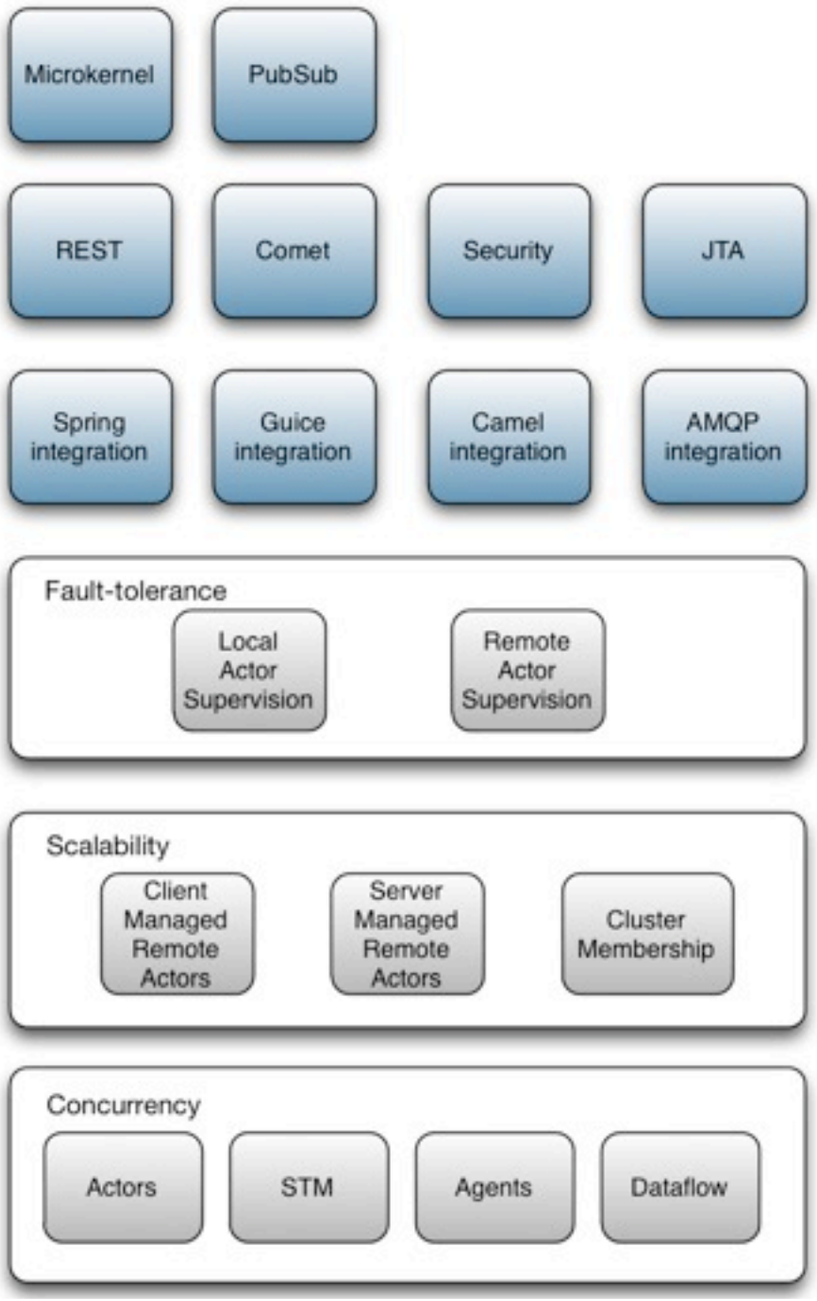
© Bob Elsdale

ARCHITECTURE



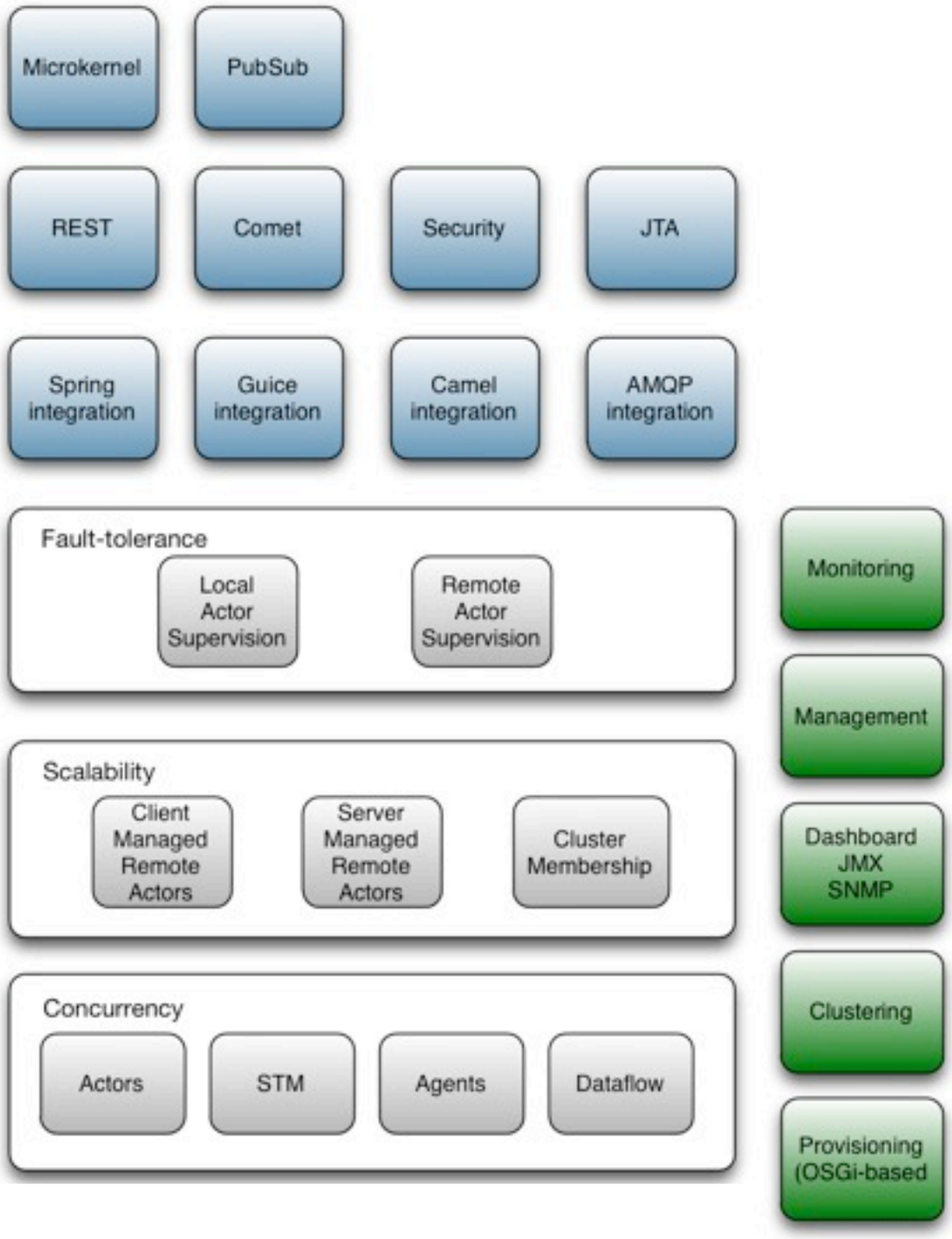
CORE SERVICES

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

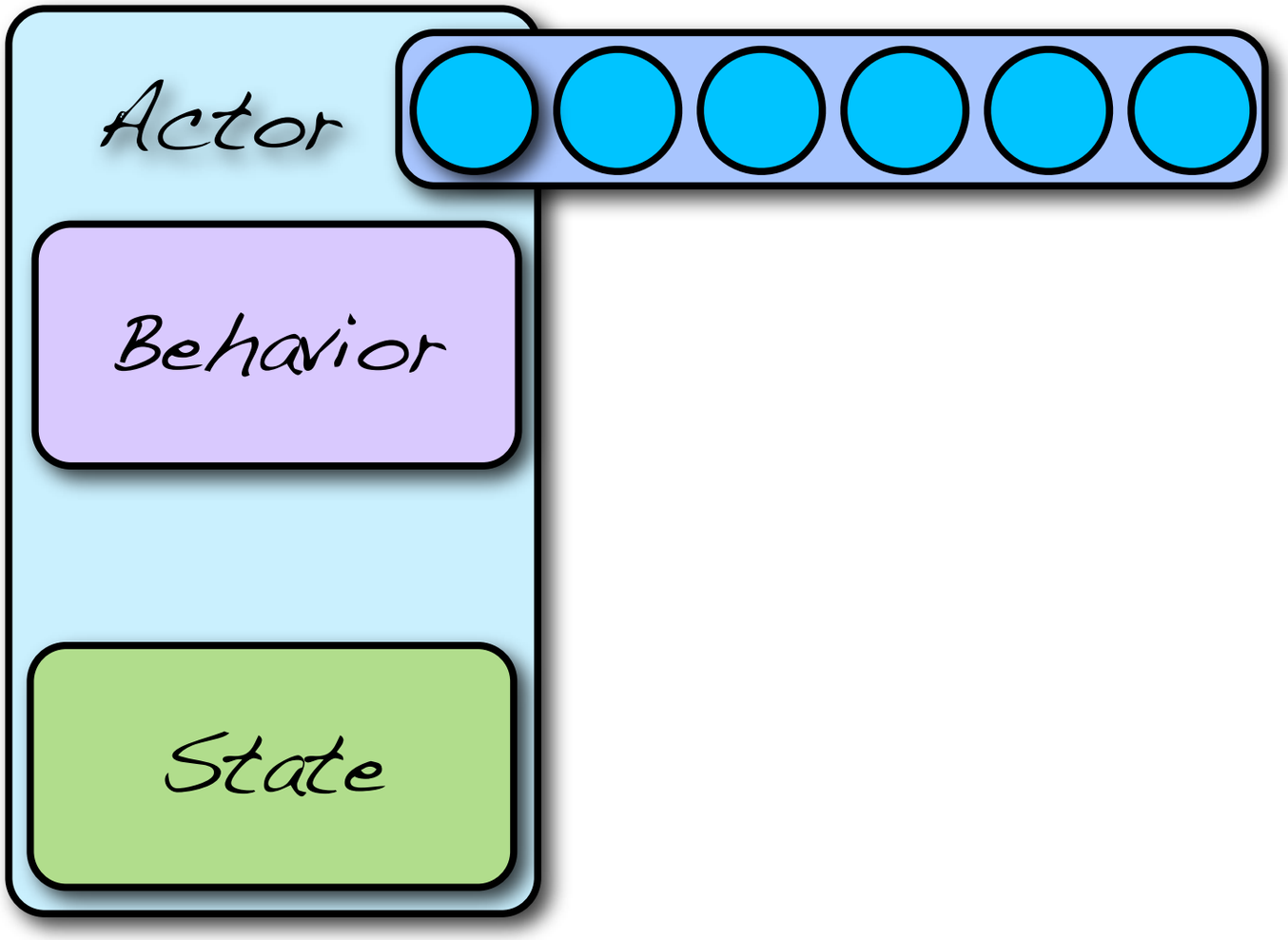
SIMULATION

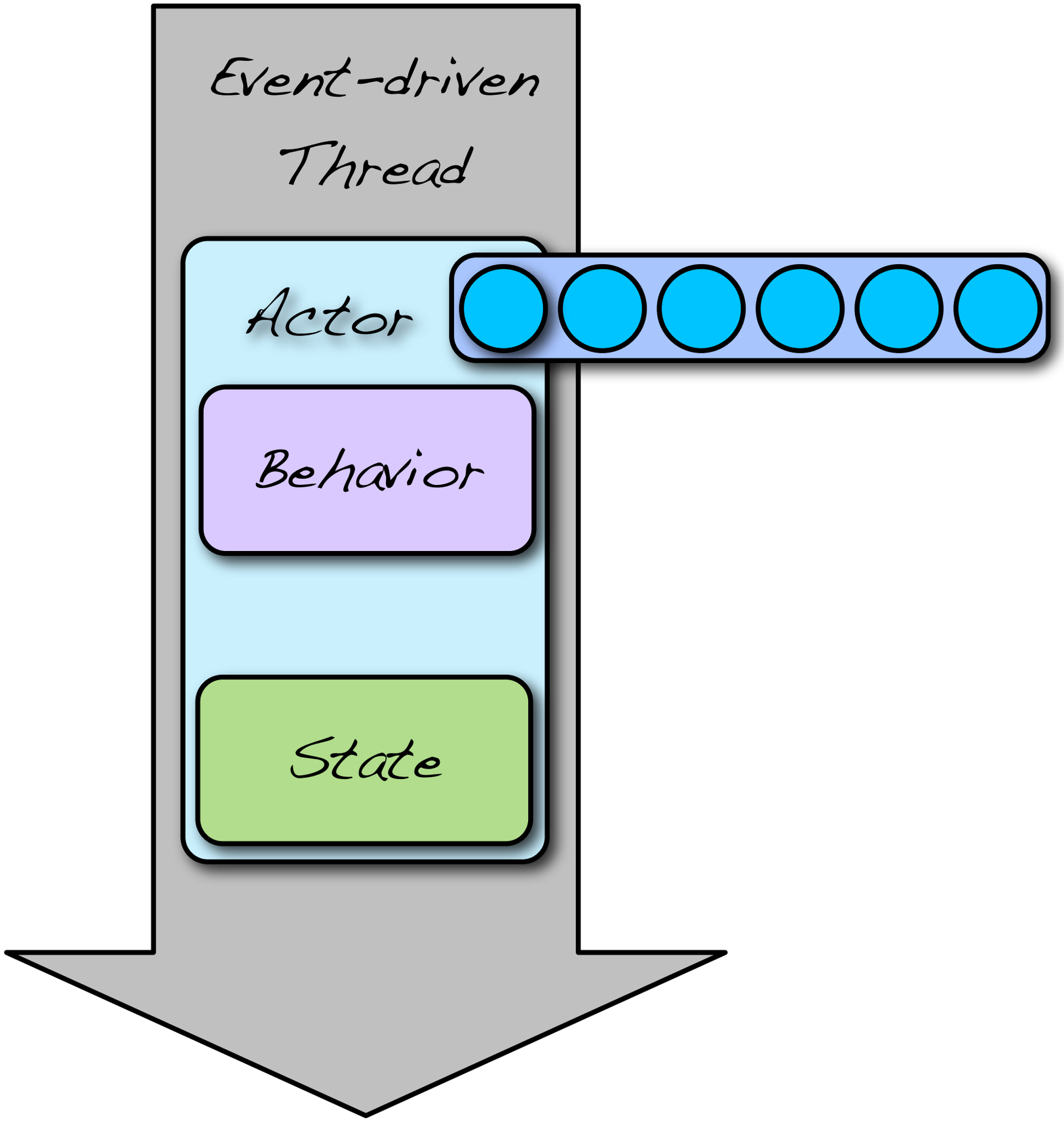
- 3D simulation engines

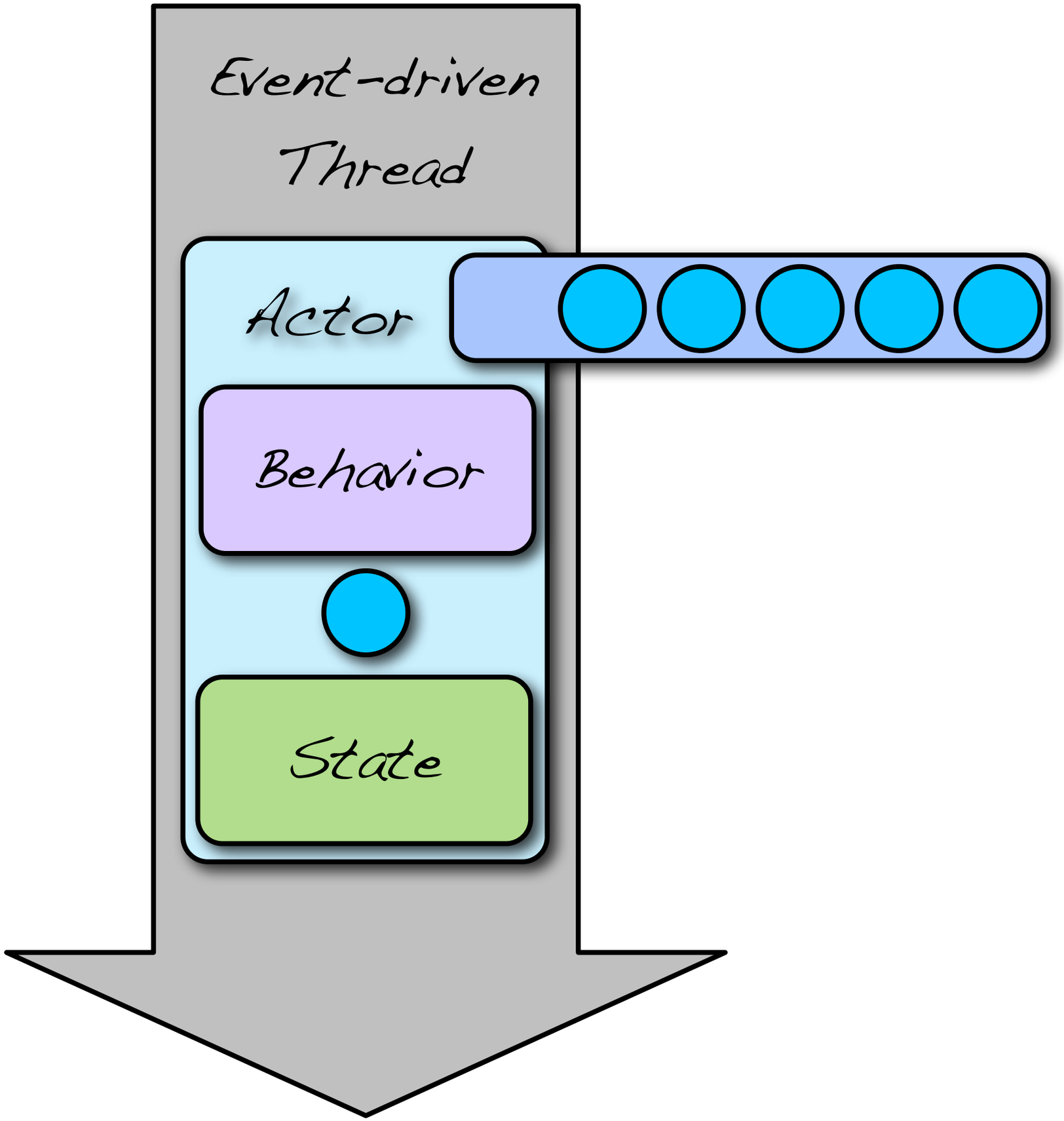
E-COMMERCE

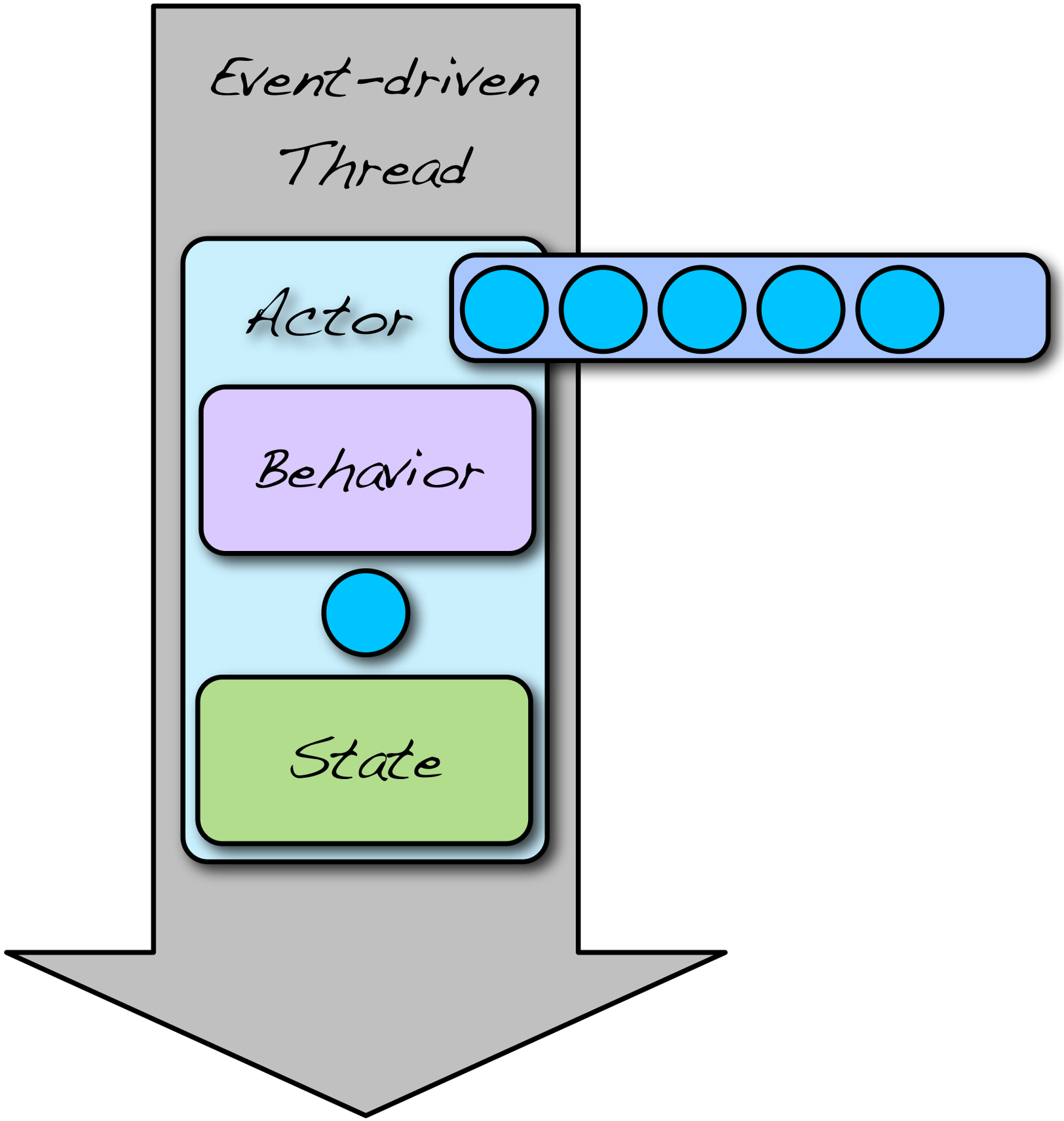
- Social media community sites

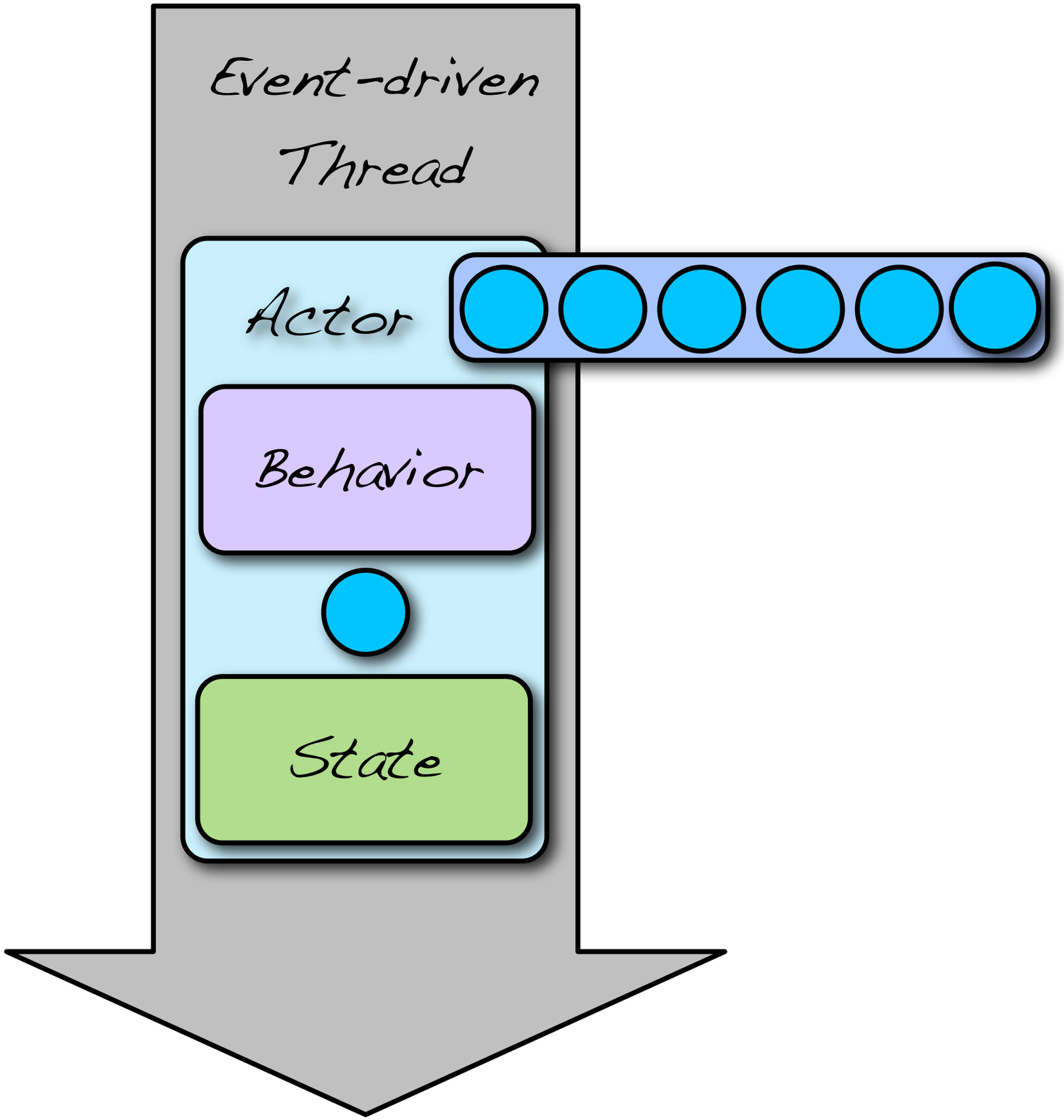
What is an Actor?

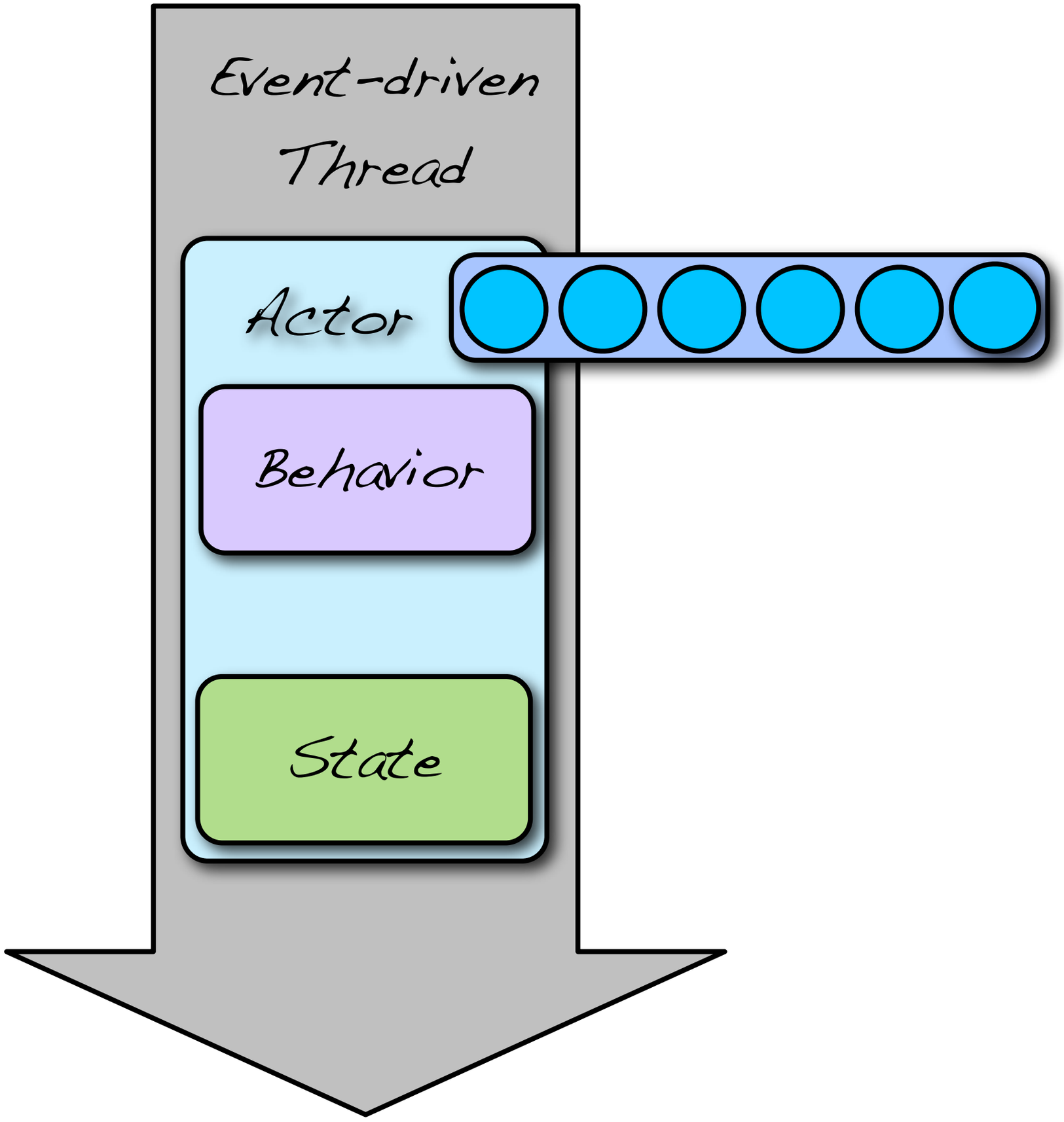

Actor
Behavior
State

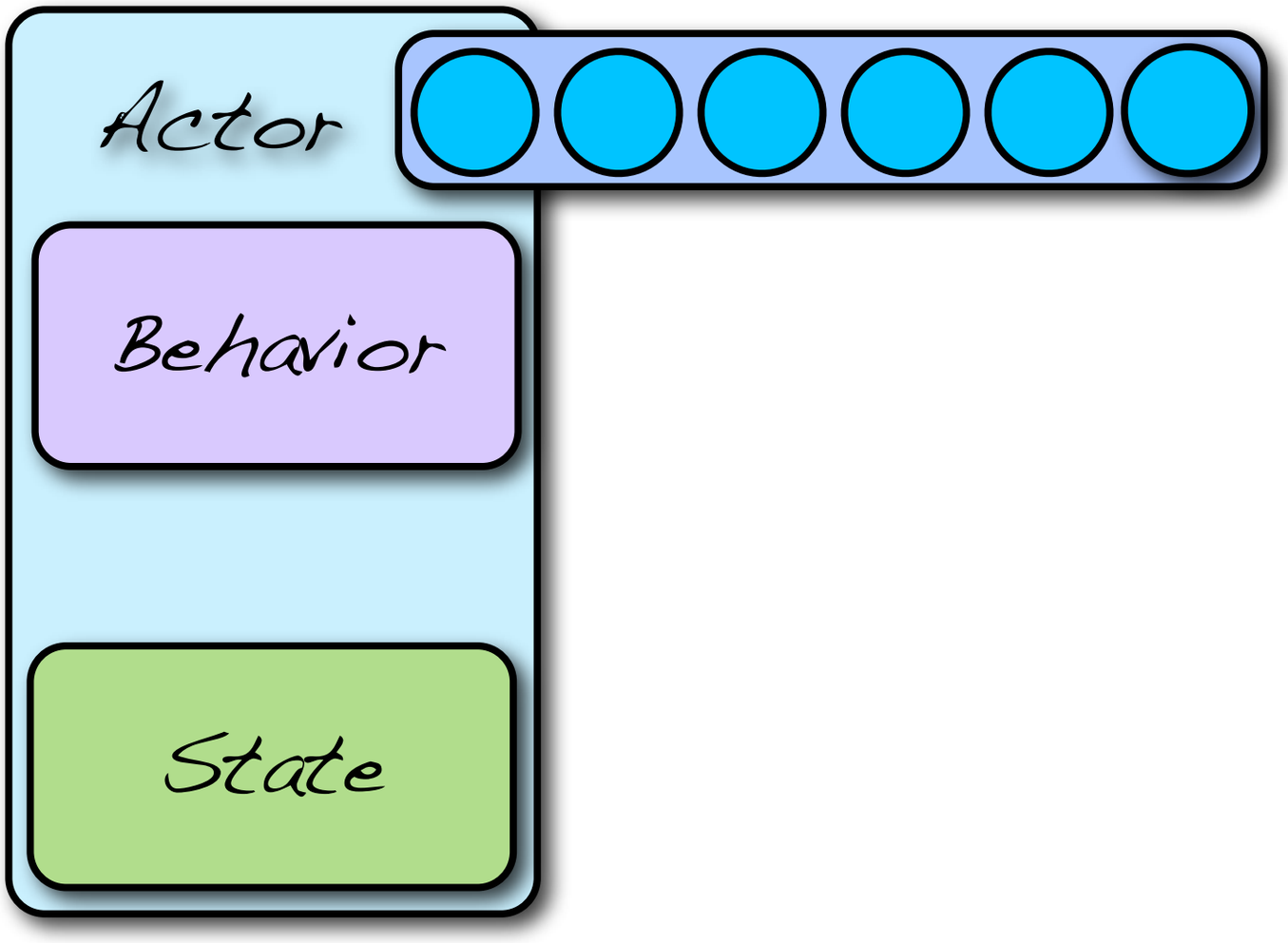

Event-driven
Thread
Actor
Behavior
State

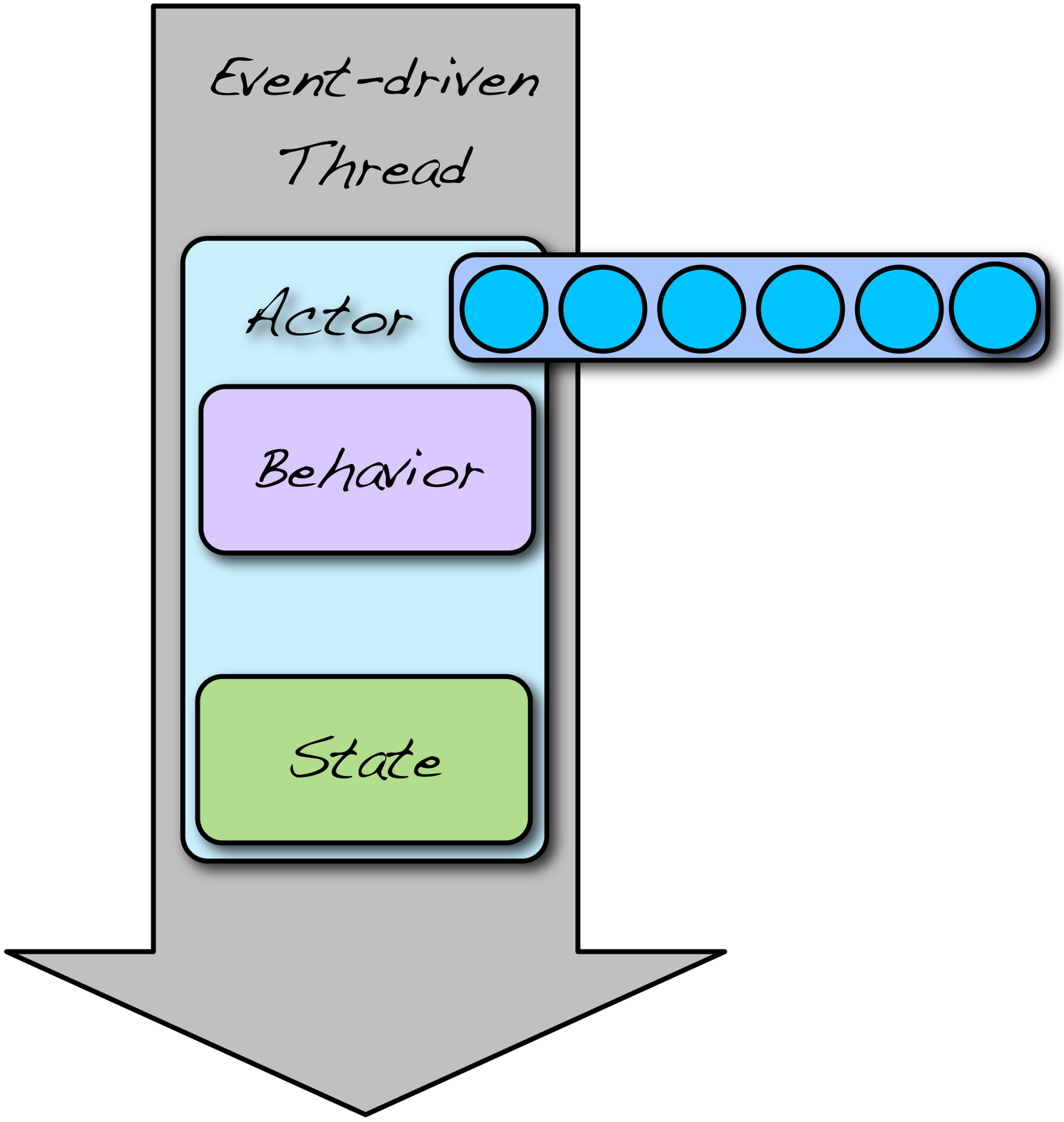

Event-driven
Thread
Actor
Behavior
State


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

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`

Start actors

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

Stop actors

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

Send: !

```
counter ! Tick
```

fire-forget

Send: !!!

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

returns the Future directly

Future

```
val future1, future2, future3 =
  Future.empty[String]

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

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

future1 receive {
  case Foo(bar) => “foo”
}.await.result
```

Future

```
val f1 = Futures.future(callable)

val f2 = Futures.firstCompletedOf(futures)
val f3 = Futures.reduce(futures)((x, y) => ..)
val f4 = Futures.fold(zero)(futures)((x, y) => ...)
```

Future

```
val future1 = for {
  a: Int    <- actor !!! "Hello" // returns 5
  b: String <- actor !!! a       // returns "10"
  c: String <- actor !!! 7       // returns "14"
} yield b + "-" + c

val future2 = for {
  a <- actor !!! Req("Hello") collect { case Res(x: Int)    => x }
  b <- actor !!! Req(a)       collect { case Res(x: String) => x }
  c <- actor !!! Req(7)       collect { case Res(x: String) => x }
} yield b + "-" + c
```

Dataflow

```
import Future.flow

val x, y, z = Promise[Int]()

flow {
  z << x() + y()
  println("z = " + z())
}
flow { x << 40 }
flow { y << 2 }
```

Send: !!

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

uses Future under the hood and blocks until timeout or completion

Reply

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

HotSwap

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

HotSwap

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

HotSwap

```
self.unbecome()
```

Set dispatcher

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

  ...
}

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

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 in Akka 1.0

Remote Actors

Client-managed

Server-managed

Problem

Deployment (local vs remote) is a dev decision

We get a fixed and hard-coded topology

Can't change it dynamically and adaptively

Needs to be a

deployment & runtime decision

Clustered Actors

(in development for upcoming Akka 2.0)

Address

```
val actor = actorOf[MyActor](“my-service”)
```

Bind the actor to a virtual address

Deployment

- Actor address is virtual and decoupled from how it is deployed
- If no deployment configuration exists then actor is deployed as local
- The same system can be configured as distributed without code change (even change at runtime)

- Write as local but deploy as distributed in the cloud without code change
- Allows runtime to dynamically and adaptively change topology

Deployment configuration

```
akka {
  actor {
    deployment {
      my-service {
        router = "least-cpu"
        clustered {
          home        = "node:test-node-1"
          replicas    = 3
          stateless = on
        }
      }
    }
  }
}
```

Deployment configuration

```
akka {
  actor {
    deployment {
      my-service {
        router = "least-cpu"
        clustered {
          home        = "node:test-node-1"
          replicas    = 3
          stateless = on
        }
      }
    }
  }
}
```

Address

Deployment configuration

```
akka {
  actor {
    deployment {
      my-service {
        router = "least-cpu"
        clustered {
          home         = "node:test-node-1"
          replicas     = 3
          stateless = on
        }
      }
    }
  }
}
```

Address

Type of load-balancing

Deployment configuration

```
akka {
  actor {
    deployment {
      my-service {
        router = "least-cpu"
        clustered {
          home     = "node:test-node-1"
          replicas = 3
          stateless = on
        }
      }
    }
  }
}
```

Address

Type of load-balancing

Clustered or Local

Deployment configuration

```
akka {
  actor {
    deployment {
      my-service {
        router = "least-cpu"
        clustered {
          home     = "node:test-node-1"
          replicas = 3
          stateless = on
        }
      }
    }
  }
}
```

Address

Type of load-balancing

Clustered or Local

Home node

Deployment configuration

```
akka {
  actor {
    deployment {
      my-service {
        router = "least-cpu"
        clustered {
          home         = "node:test-node-1"
          replicas     = 3
          stateless    = on
        }
      }
    }
  }
}
```

Address

Type of load-balancing

Clustered or Local

Home node

Nr of replicas in cluster

Deployment configuration

```
akka {
  actor {
    deployment {
      my-service {
        router = "least-cpu"
        clustered {
          home         = "node:test-node-1"
          replicas     = 3
          stateless    = on
        }
      }
    }
```

Address

Type of load-balancing

Clustered or Local

Home node

Stateful or Stateless

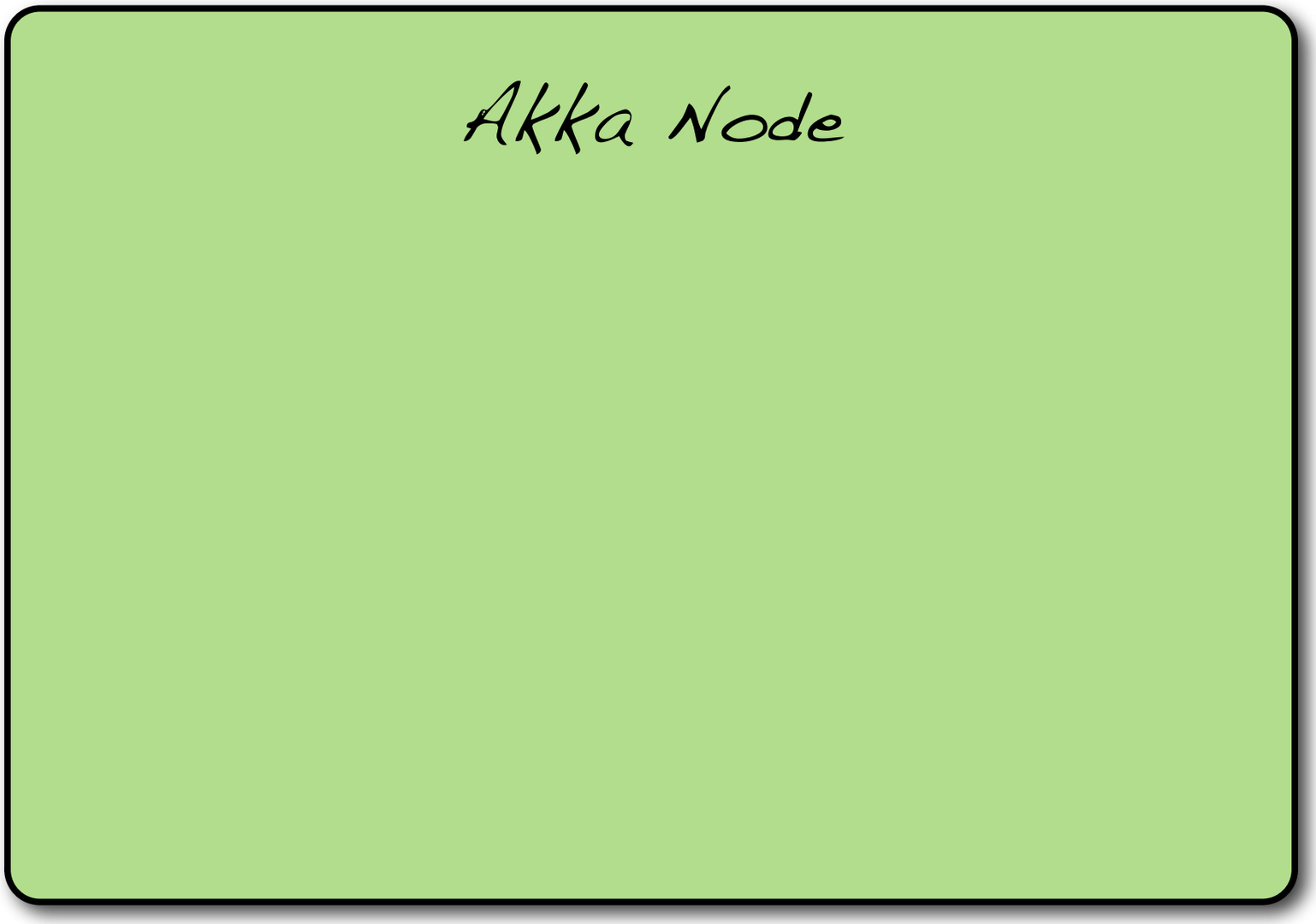
Nr of replicas in cluster

The runtime provides

- Subscription-based cluster membership service
- Highly available cluster registry for actors
- Automatic cluster-wide deployment
- 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
- Durable mailboxes - guaranteed delivery

Upcoming features

- Publish/Subscribe (broker and broker-less)
- Compute Grid (MapReduce)
- Data Grid (querying etc)
- Distributed STM (Transactional Memory)
- Event Sourcing


Akka Node

Akka Node

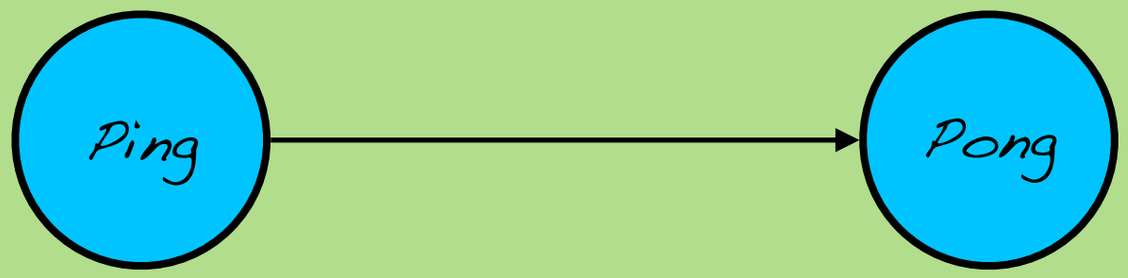
```
val ping = actorOf[Ping](“ping”)
val pong = actorOf[Pong](“pong”)

ping ! Ball(pong)
```

Akka Node

```
val ping = actorOf[Ping](“ping”)
val pong = actorOf[Pong](“pong”)

ping ! Ball(pong)
```




Akka
Cluster Node


Ping


Pong

Akka
Cluster Node

Akka
Cluster Node

Akka
Cluster Node

Ping

Pong

Akka
Cluster Node

Akka
Cluster Node

Akka
Cluster Node

Akka
Cluster Node

Akka
Cluster Node

Ping

Pong

Akka
Cluster Node

Akka
Cluster Node

Akka Cluster Node

Akka Cluster Node

```
akka {
  actor {
    deployment {
      ping {}
      pong {
        router = "round-robin"
        clustered {
          replicas  = 3
          stateless = on
        }
      }
    }
}
```

Ping

Pong

Akka Cluster Node

Akka Cluster Node

Akka Cluster Node

Akka
Cluster Node

```
akka {
  actor {
    deployment {
      ping {}
      pong {
        router = "round-robin"
        clustered {
          replicas  = 3
          stateless = on
        }
      }
    }
  }
}
```

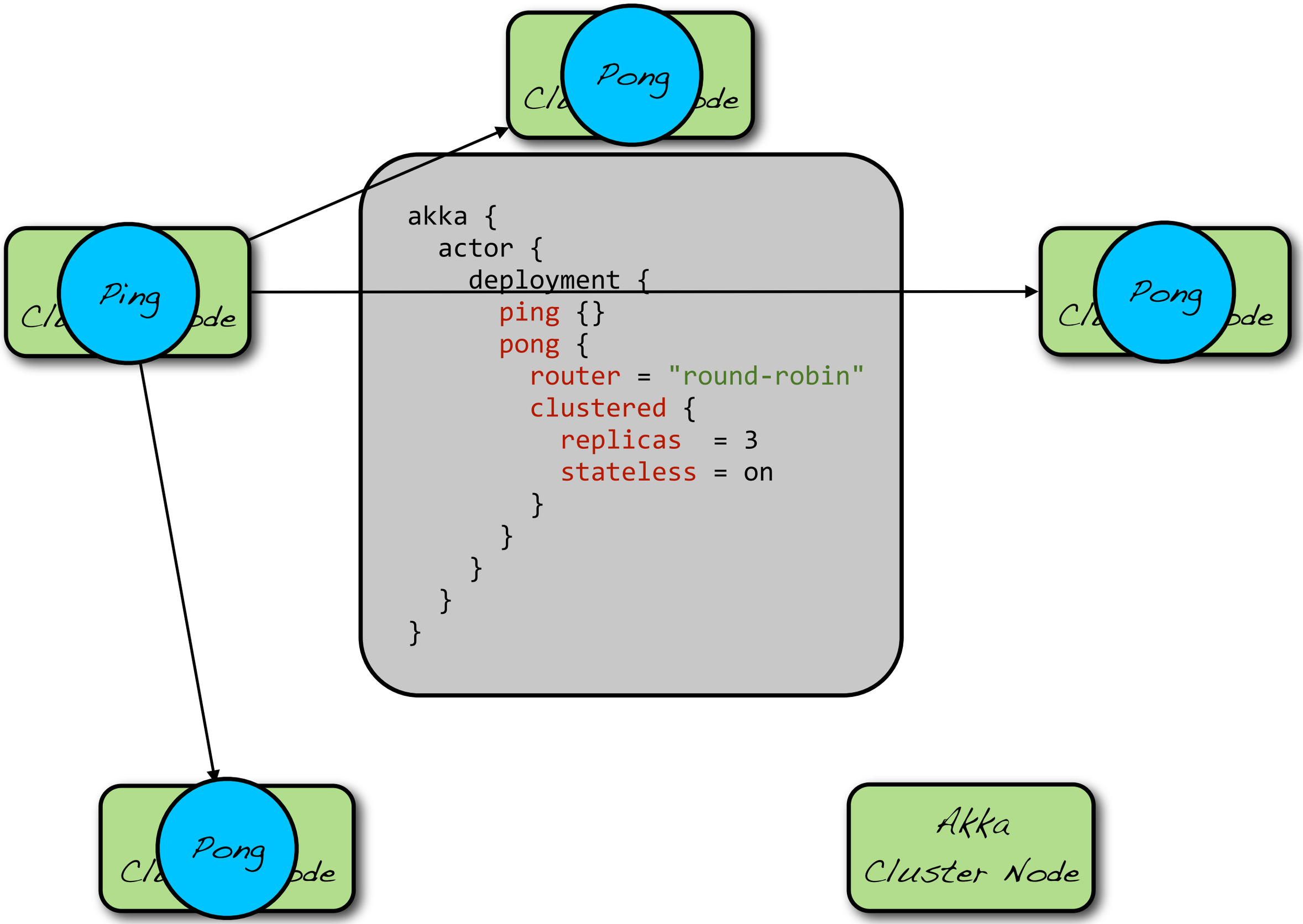
Ping

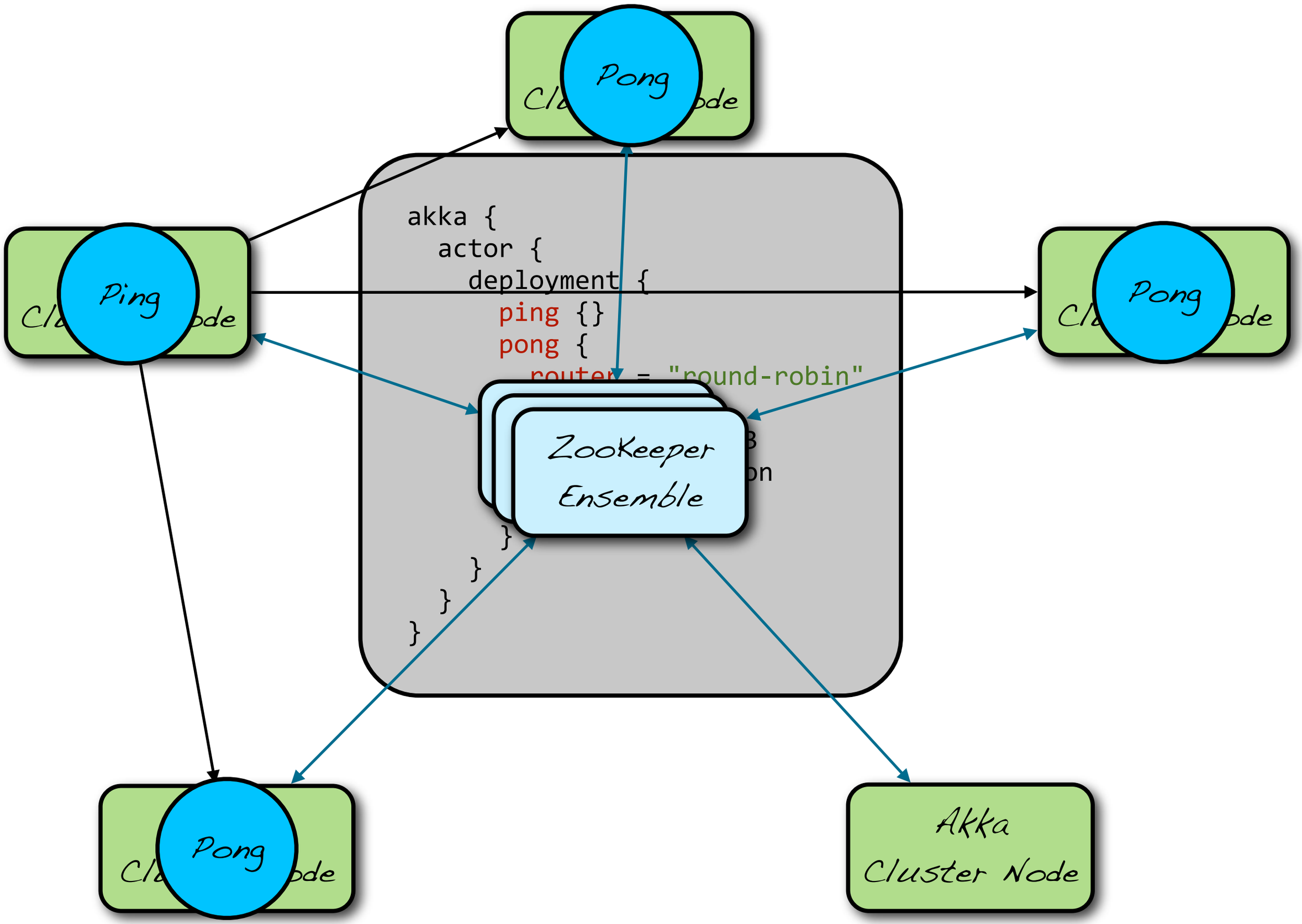
Pong

Akka
Cluster Node

Akka
Cluster Node

Akka
Cluster Node


Pong
Ping
Pong
Pong
akka {
actor {
deployment {
ping {}
pong {
router = "round-robin"
clustered {
replicas = 3
stateless = on
}
}
}
}
}
Akka
Cluster Node


Pong
Ping
Pong
akka {
actor {
deployment {
ping {}
pong {
router = "round-robin"
}
}
}
}
}
ZooKeeper
Ensemble
Pong
Akka
Cluster Node

Let it crash

fault-tolerance

The Erlang model

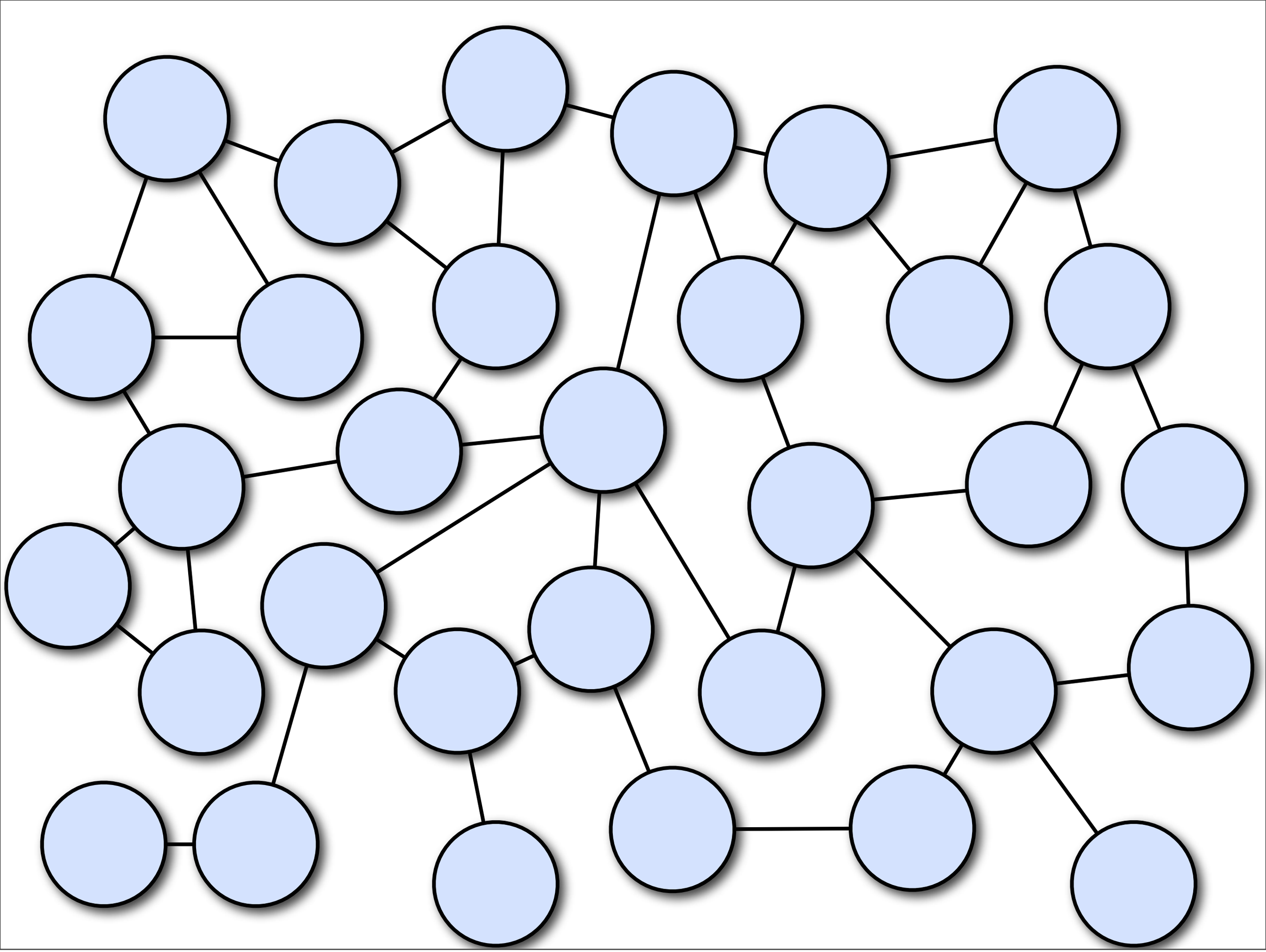
9

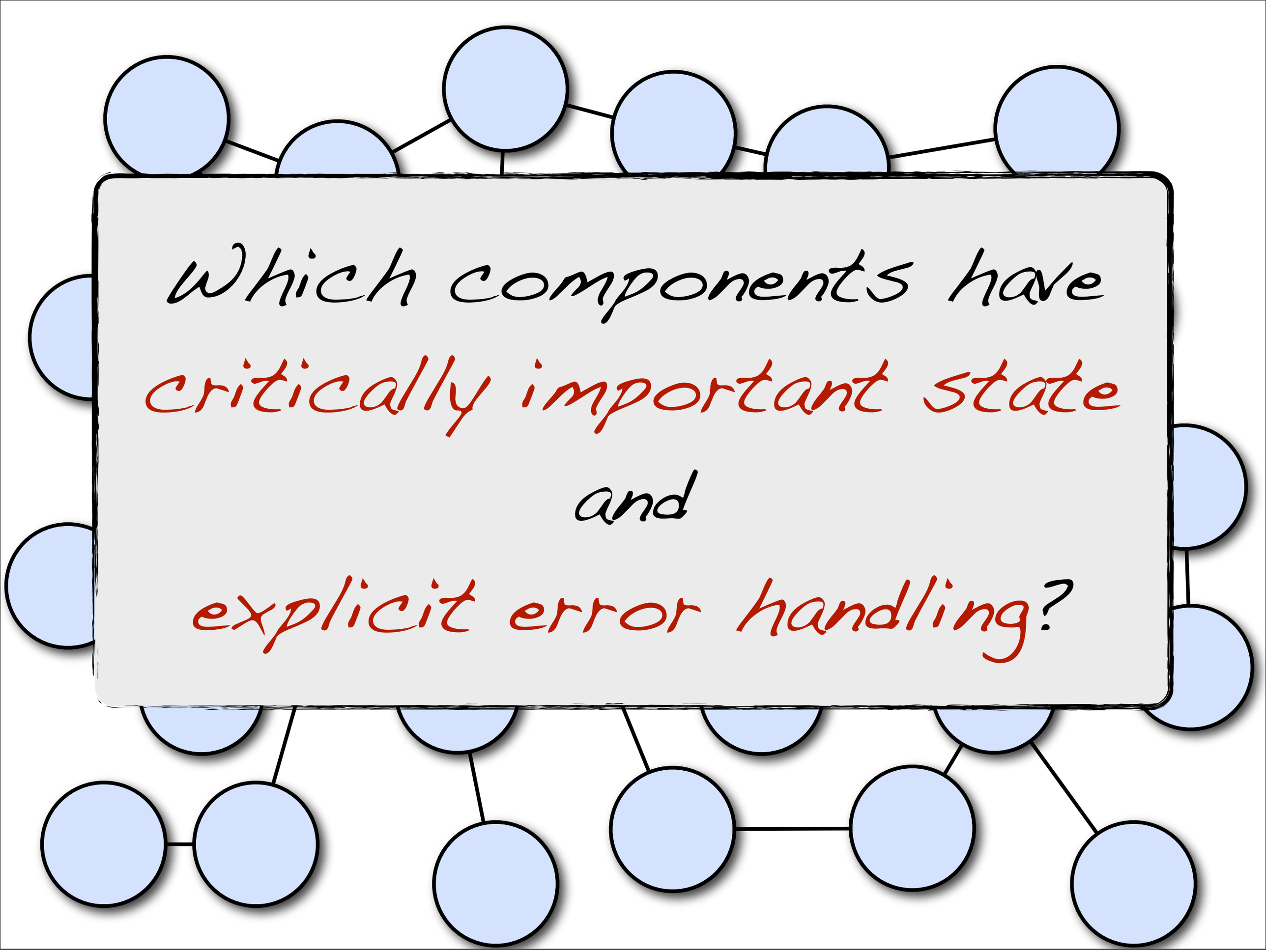
nines

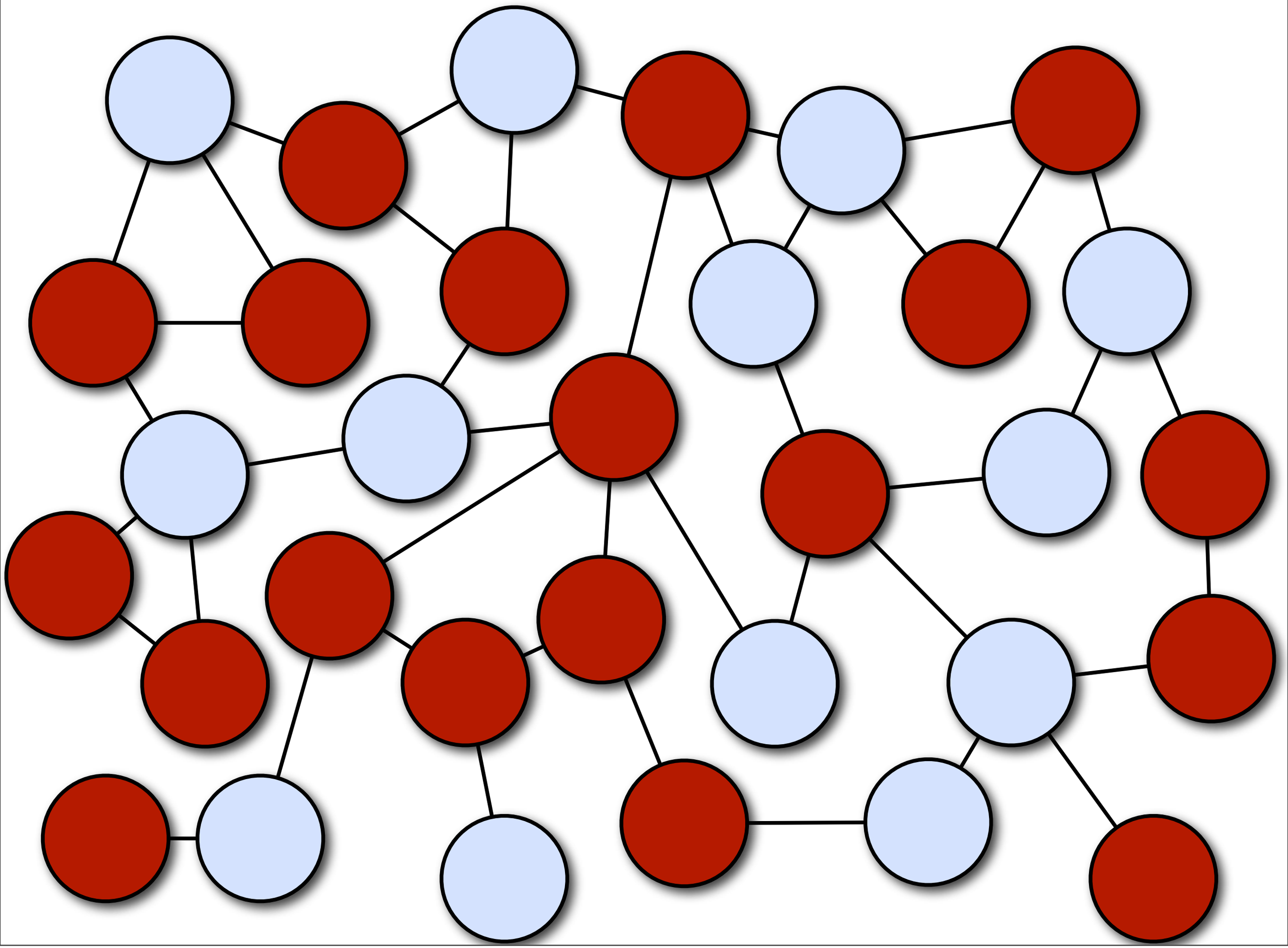
...let's take a

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

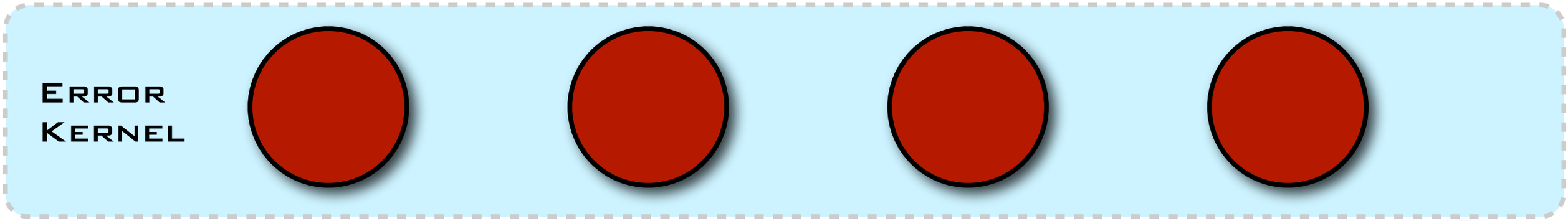
Must be

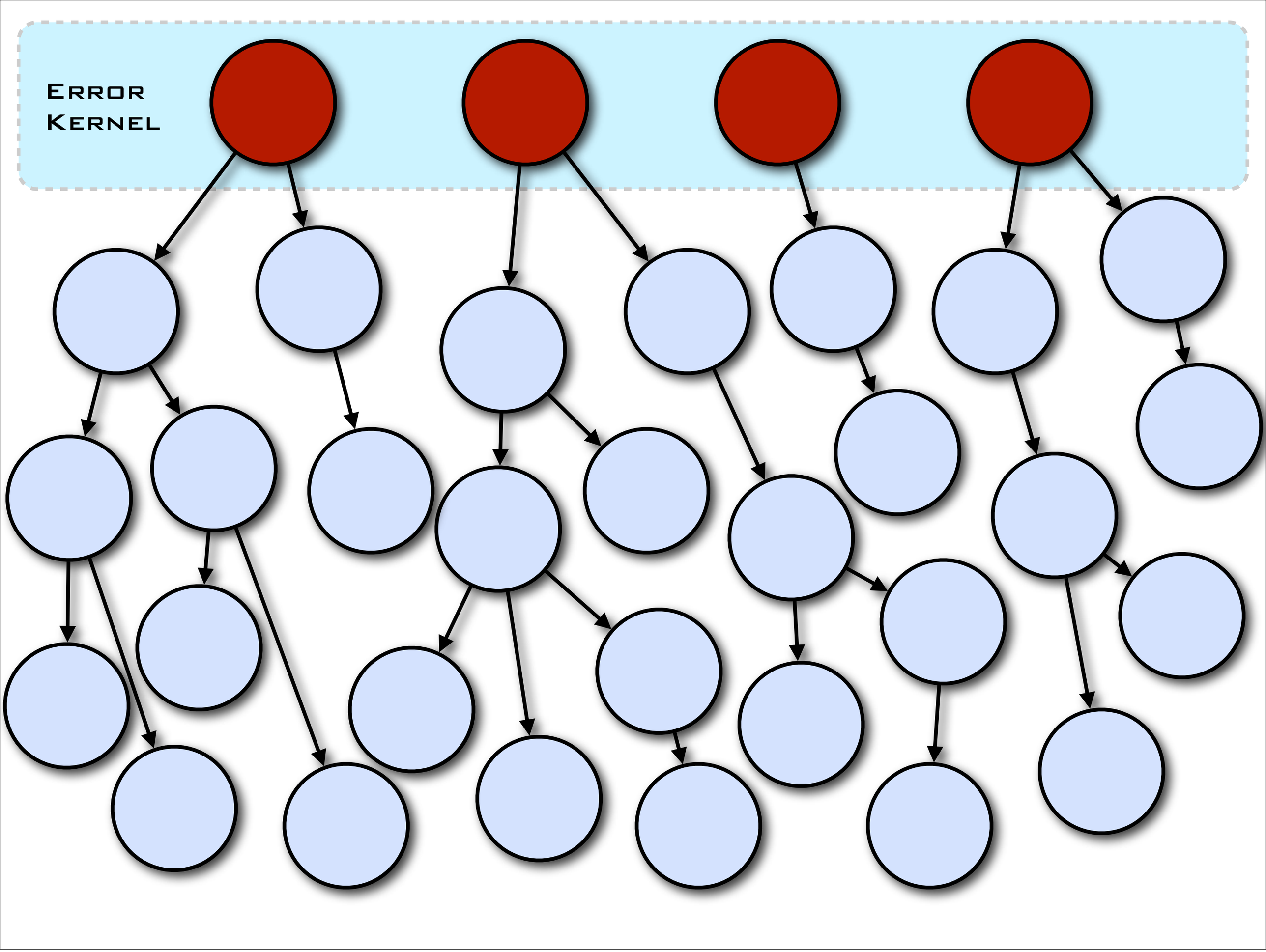
protected

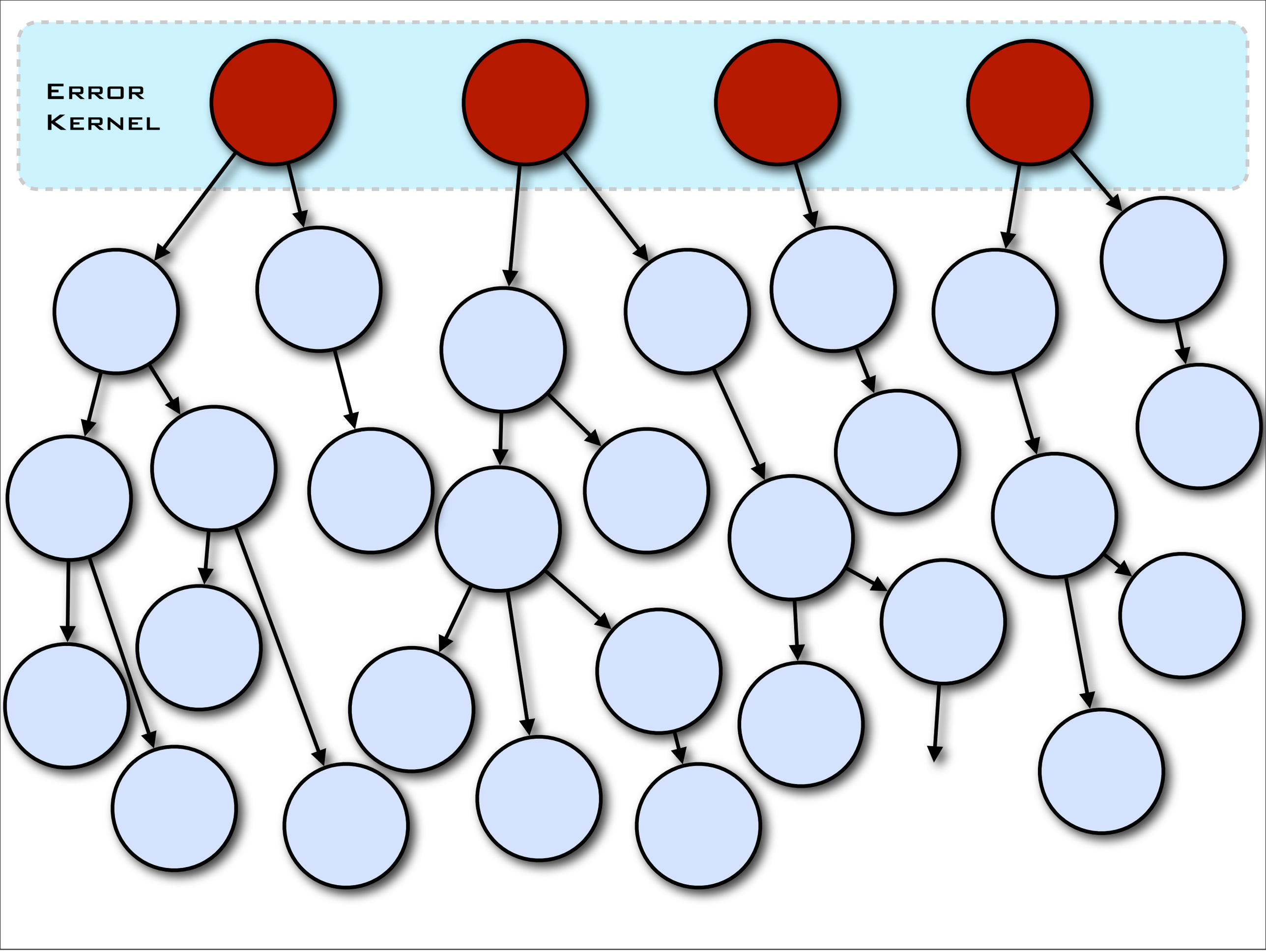
by any means

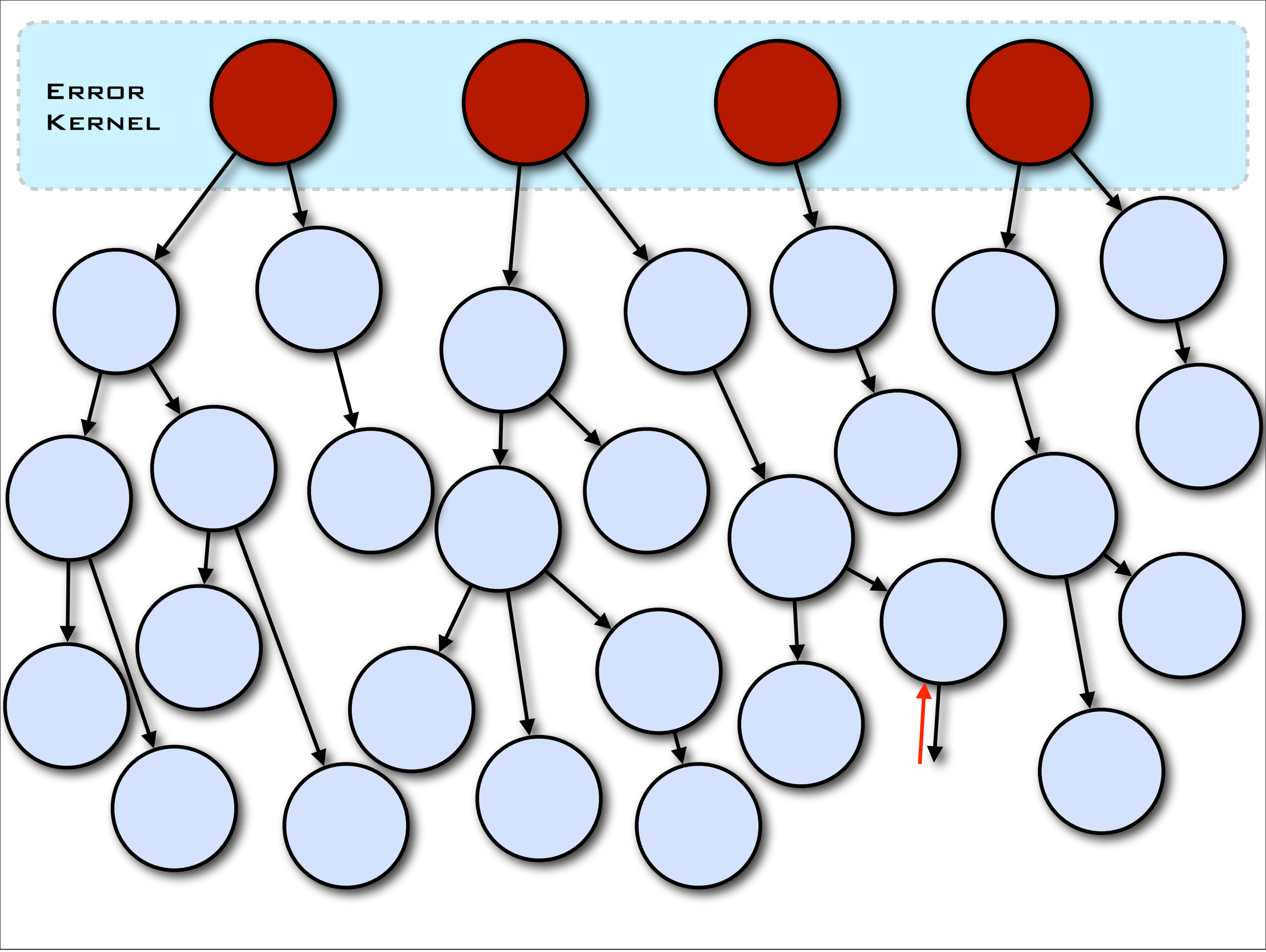
- Input *from other sources*; data that is *impossible to recompute*

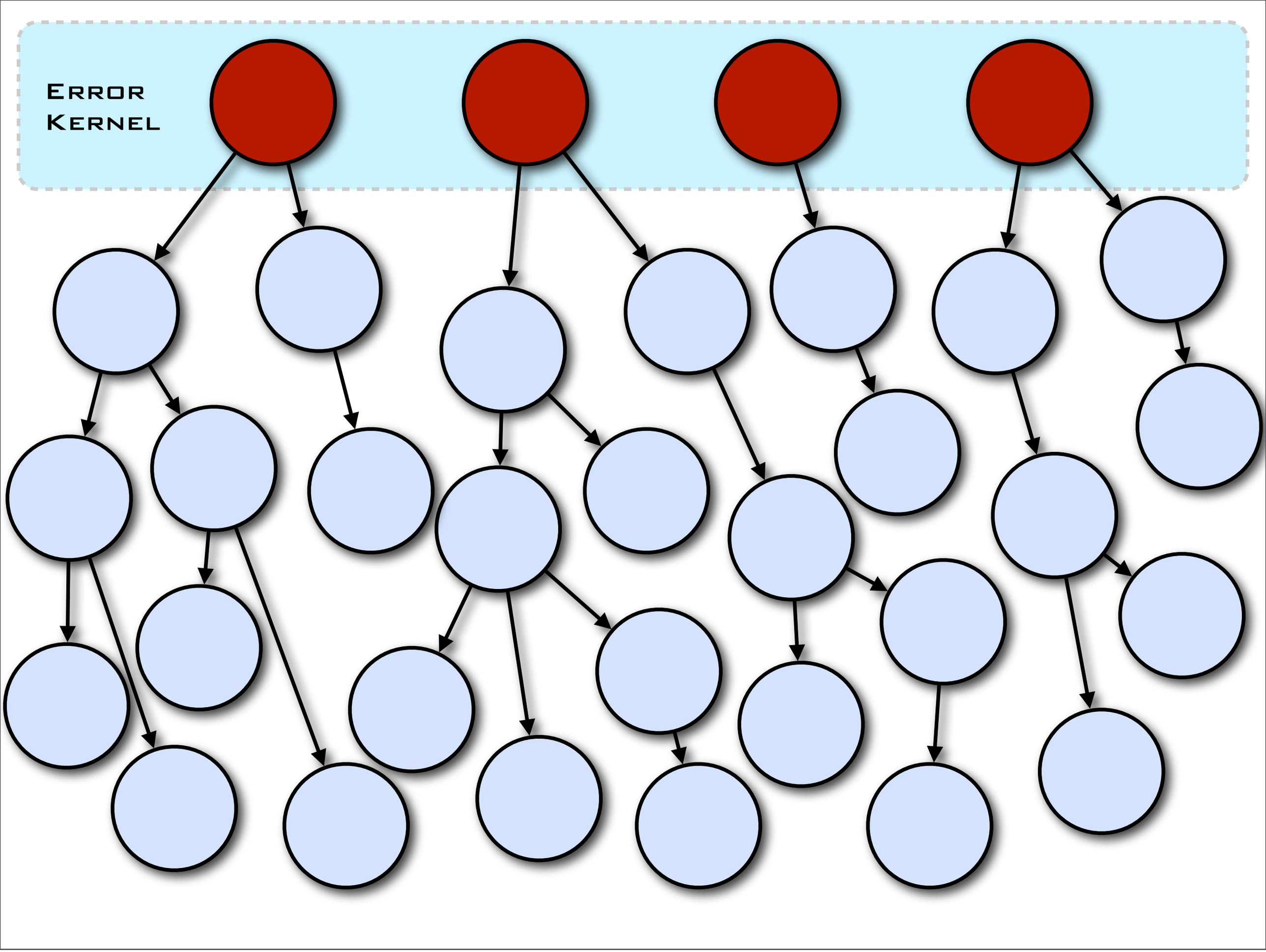
Fault-tolerant onion-layered Error Kernel

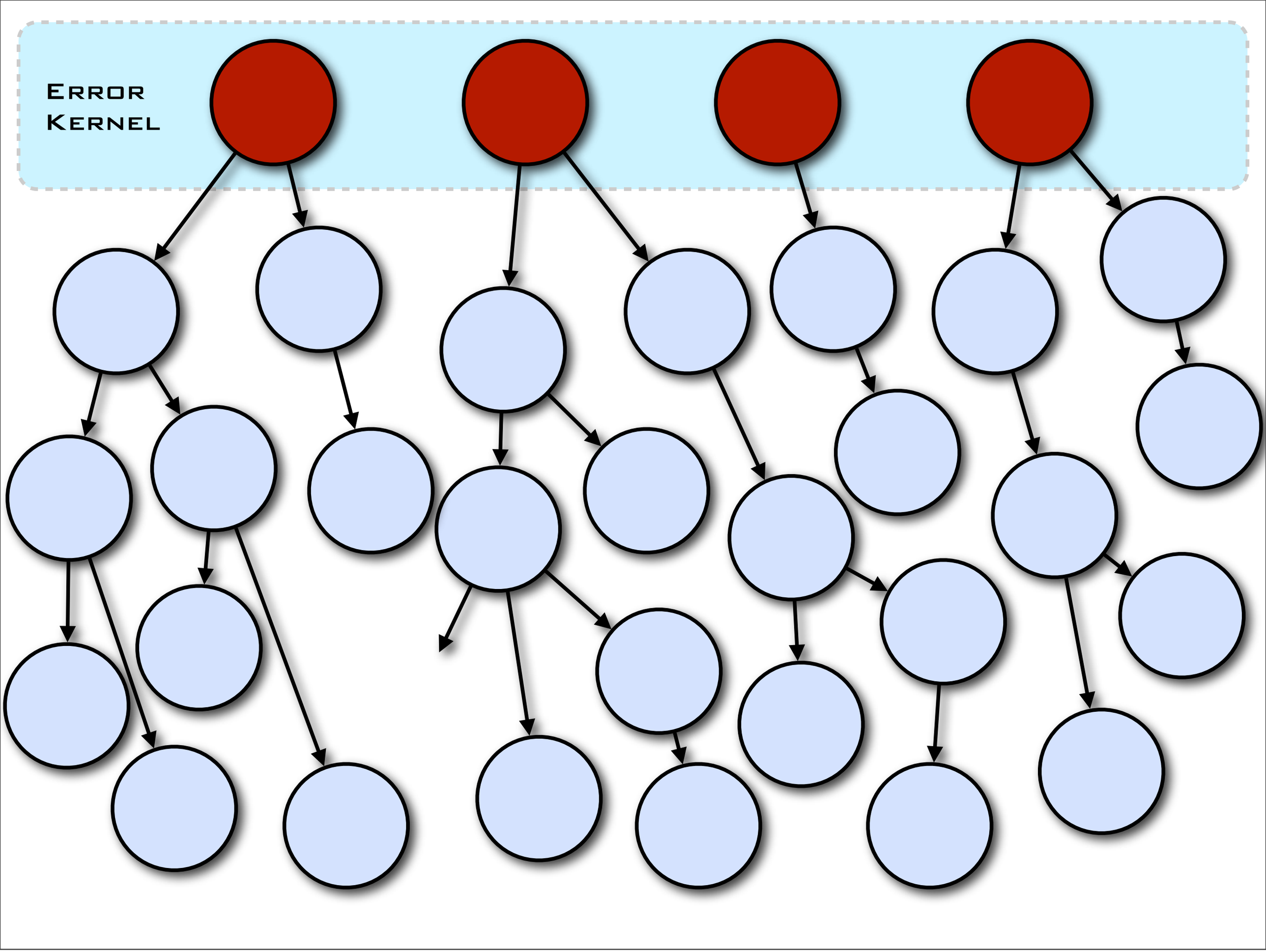

Error
Kernel

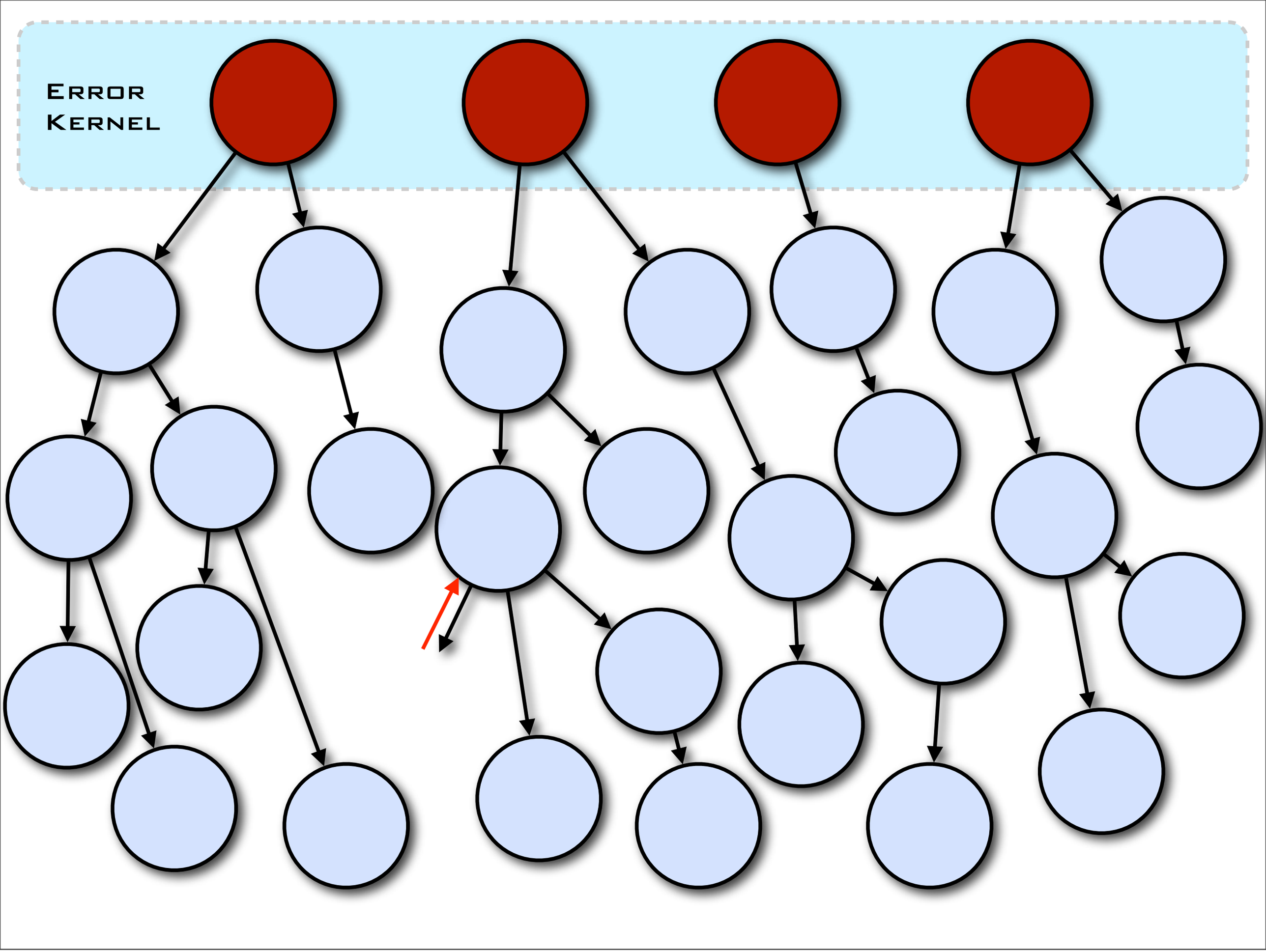

Error
Kernel

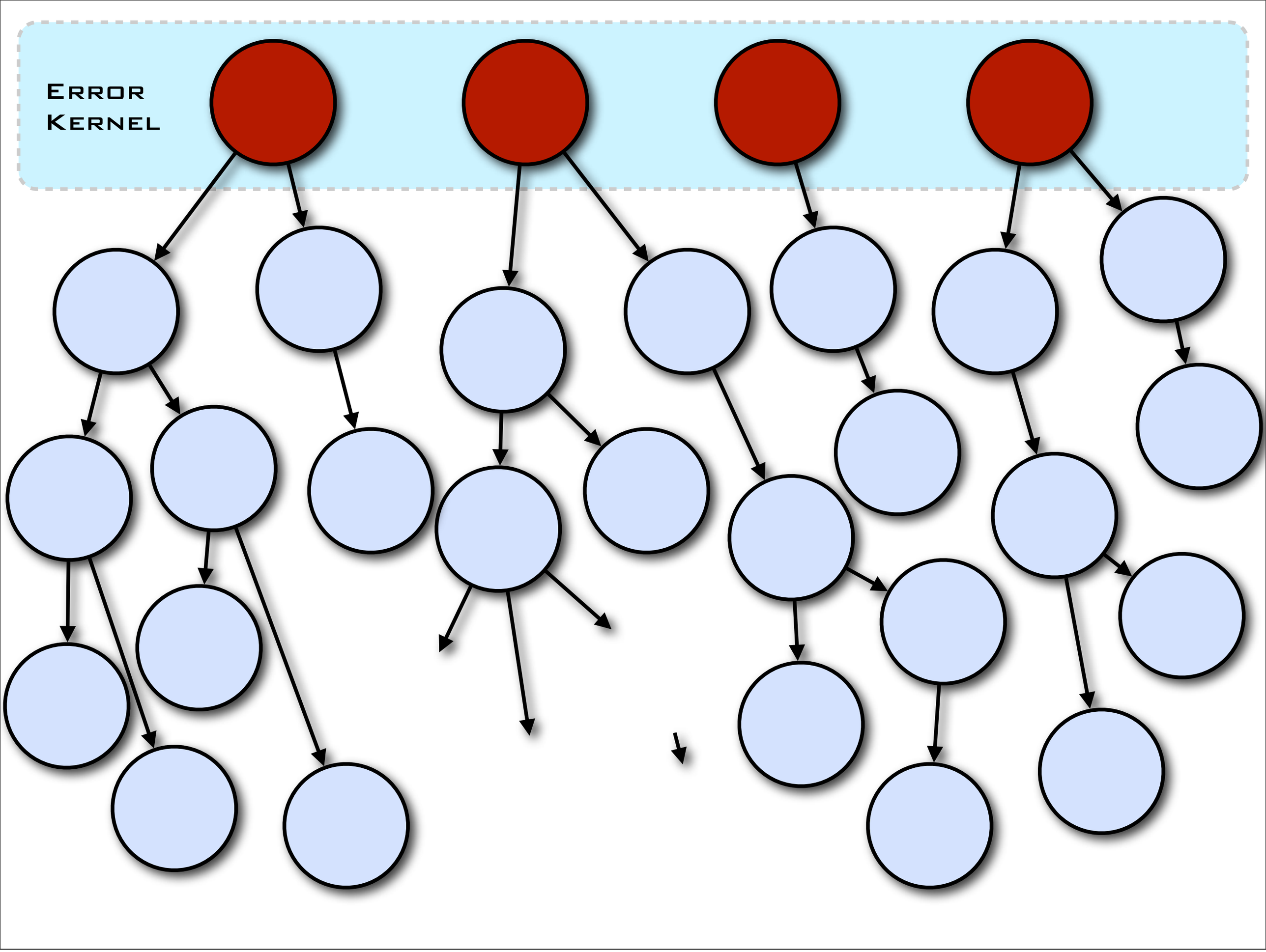

Error
Kernel

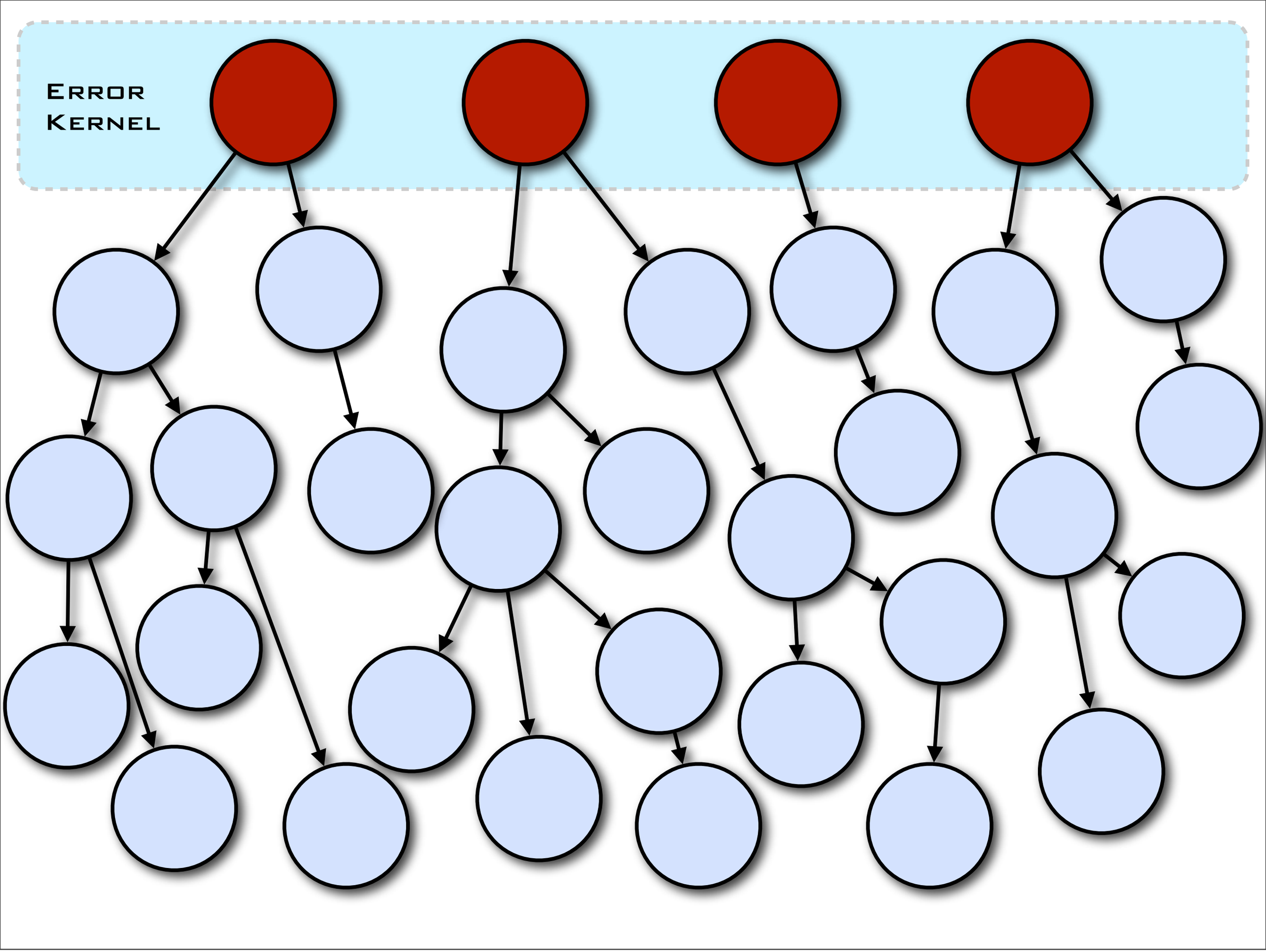

Error
Kernel

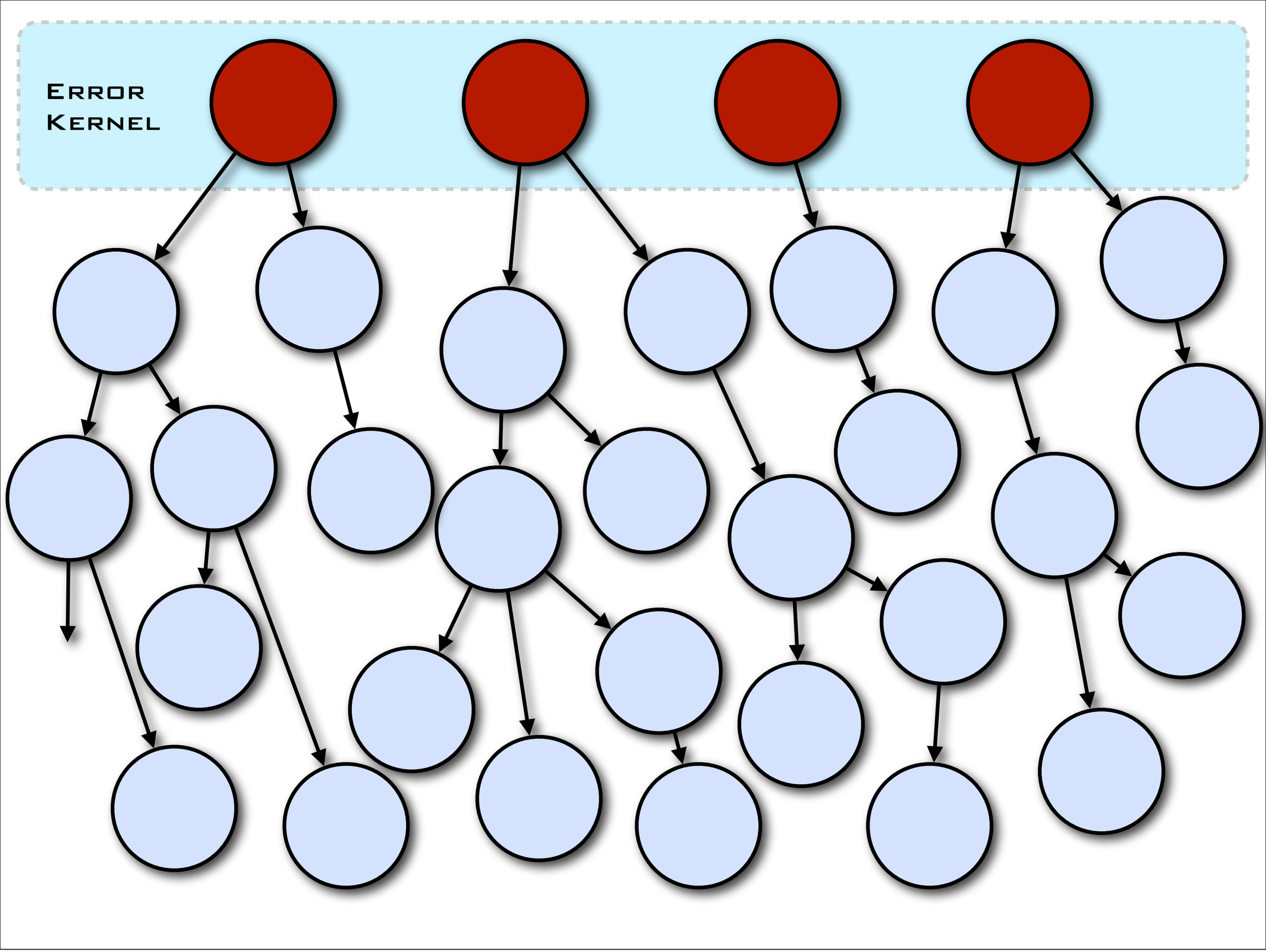

Error
Kernel

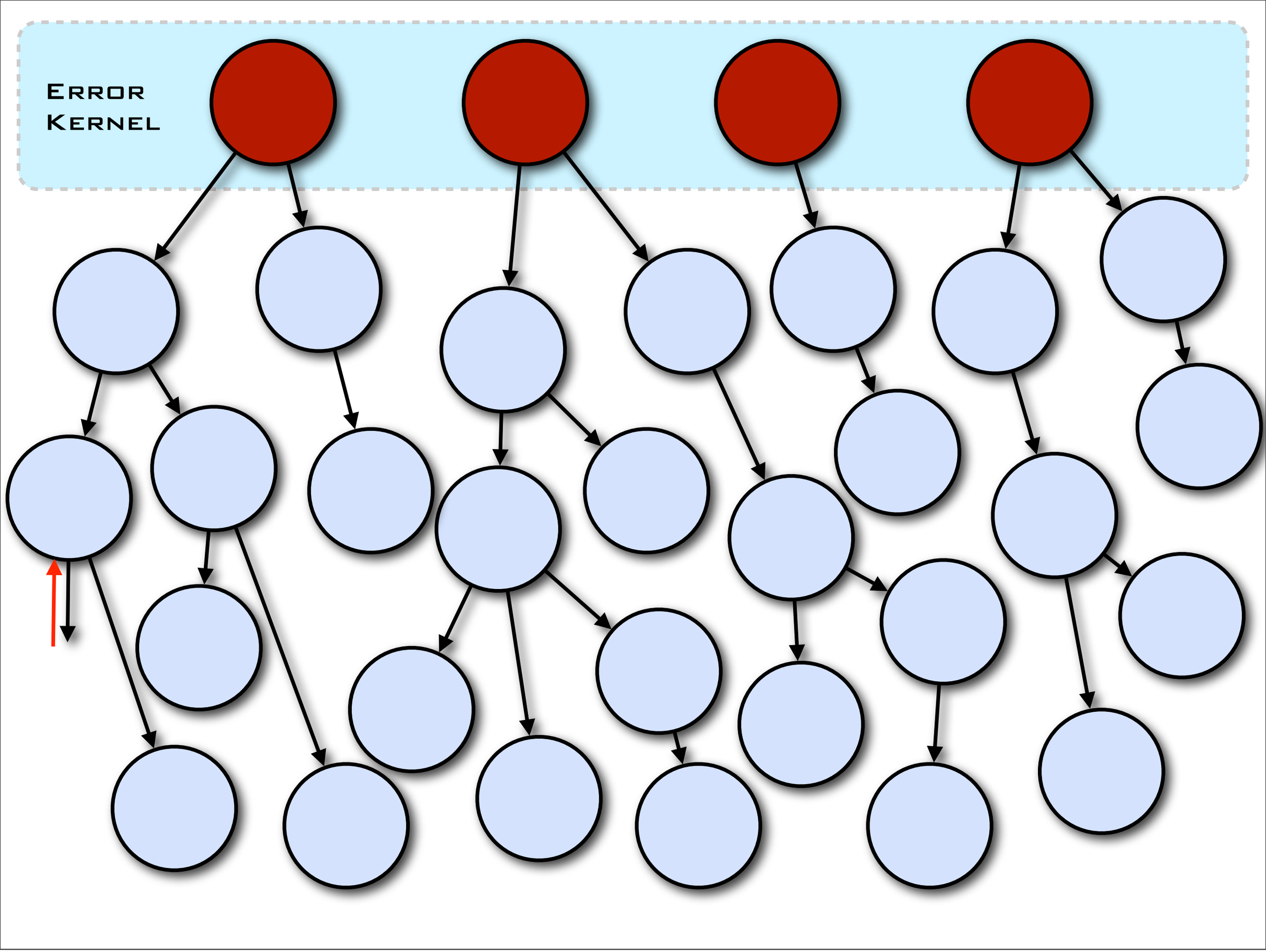

Error
Kernel

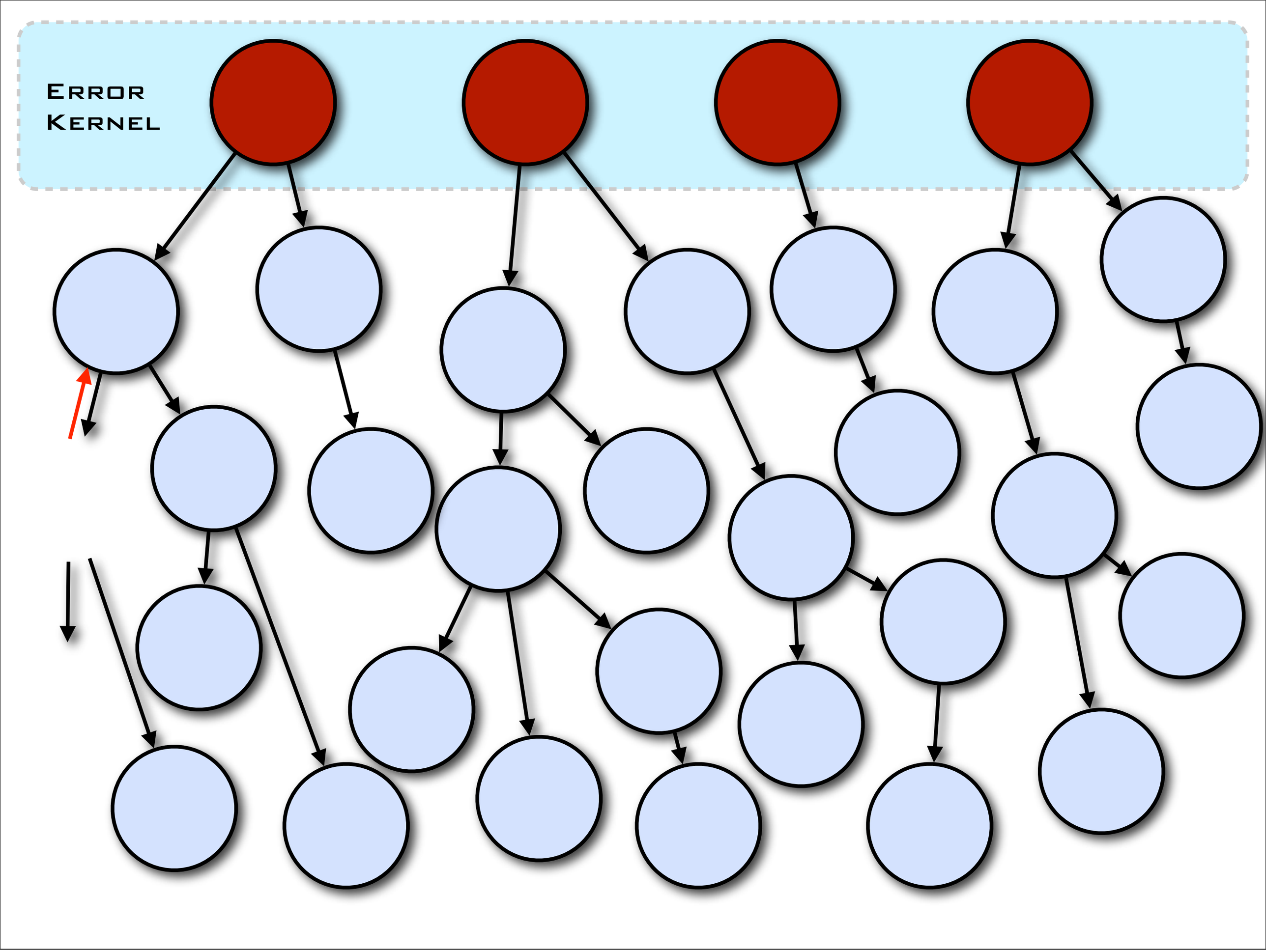

Error
Kernel

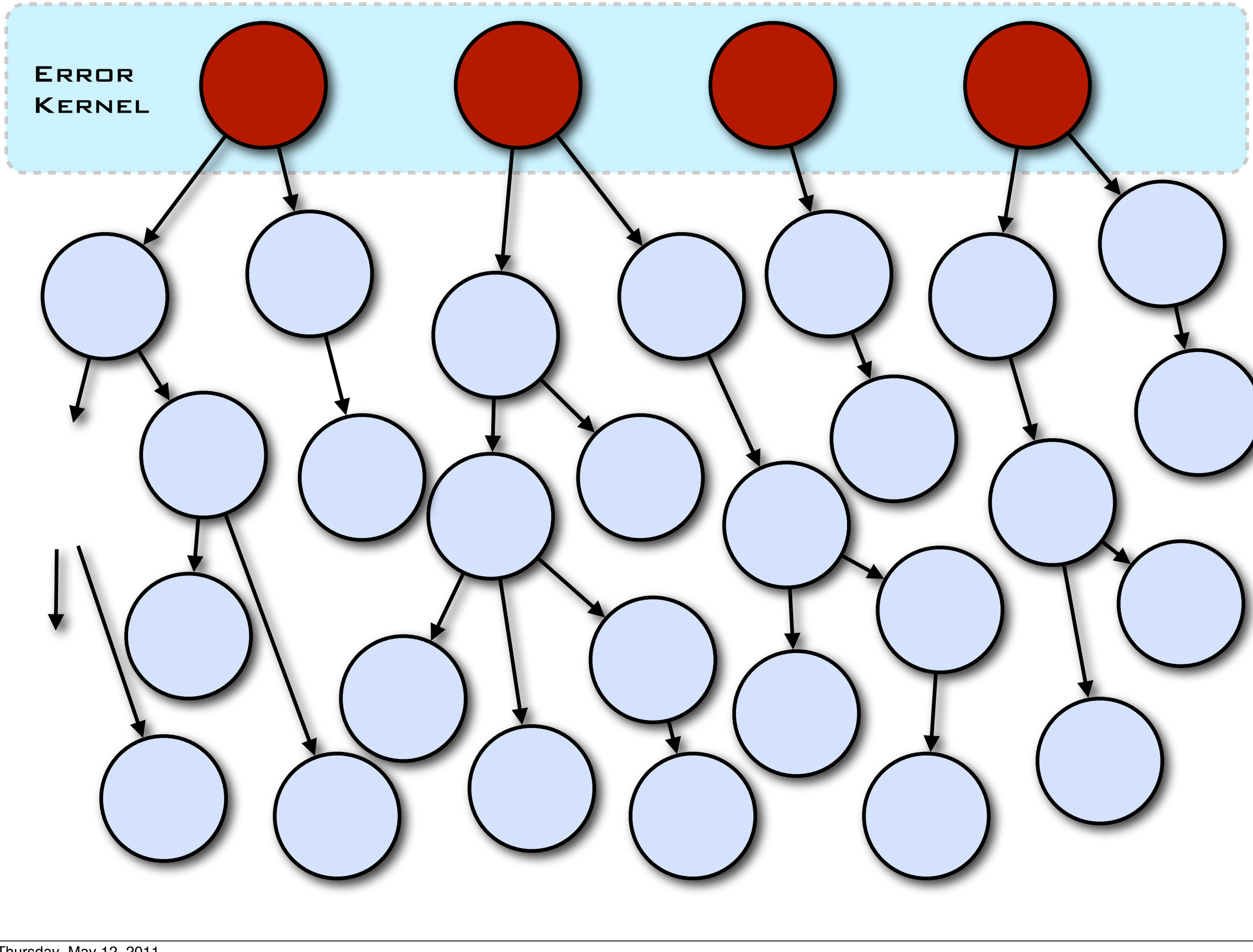

Error
Kernel

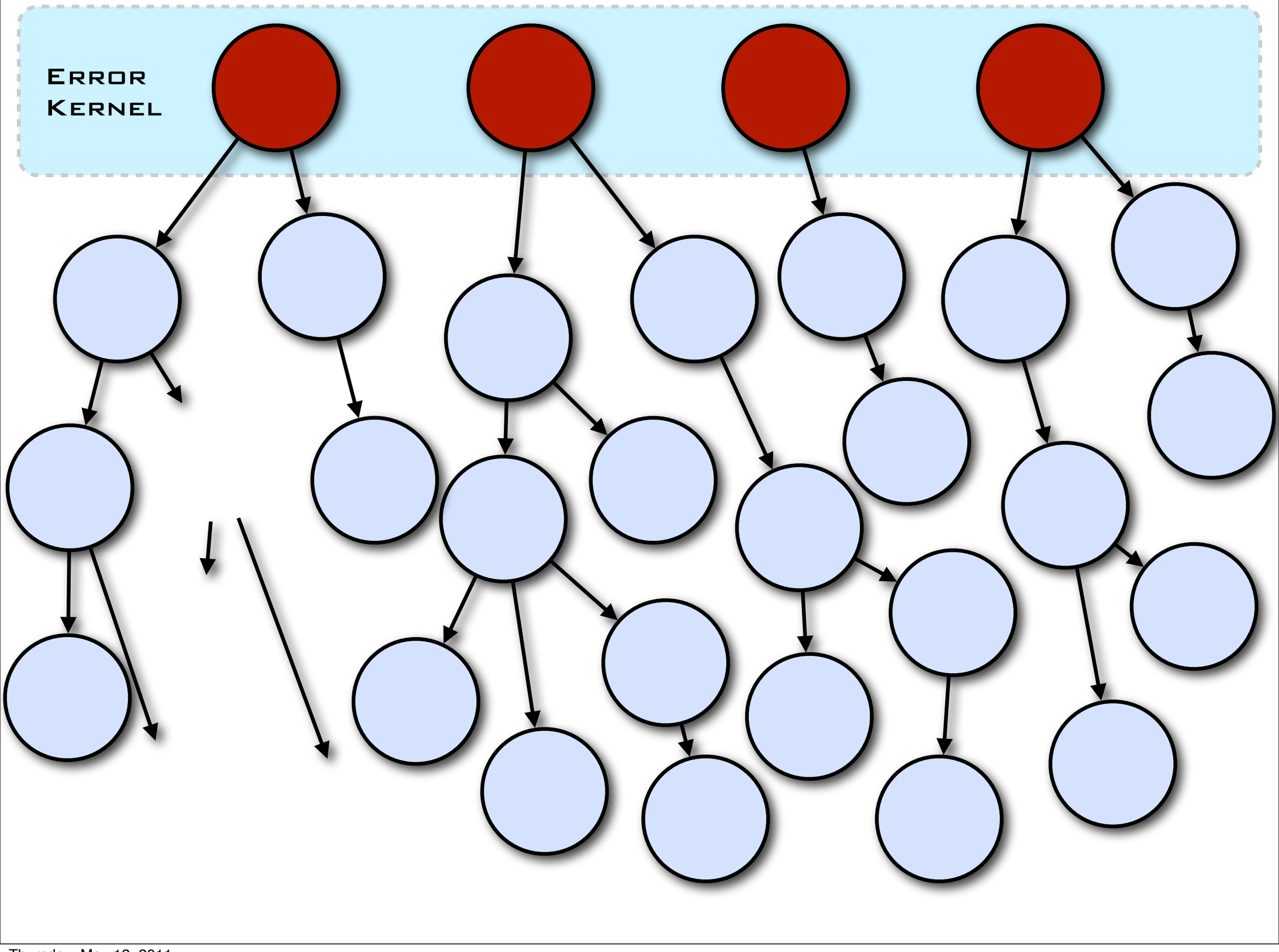

Error
Kernel


Error
Kernel

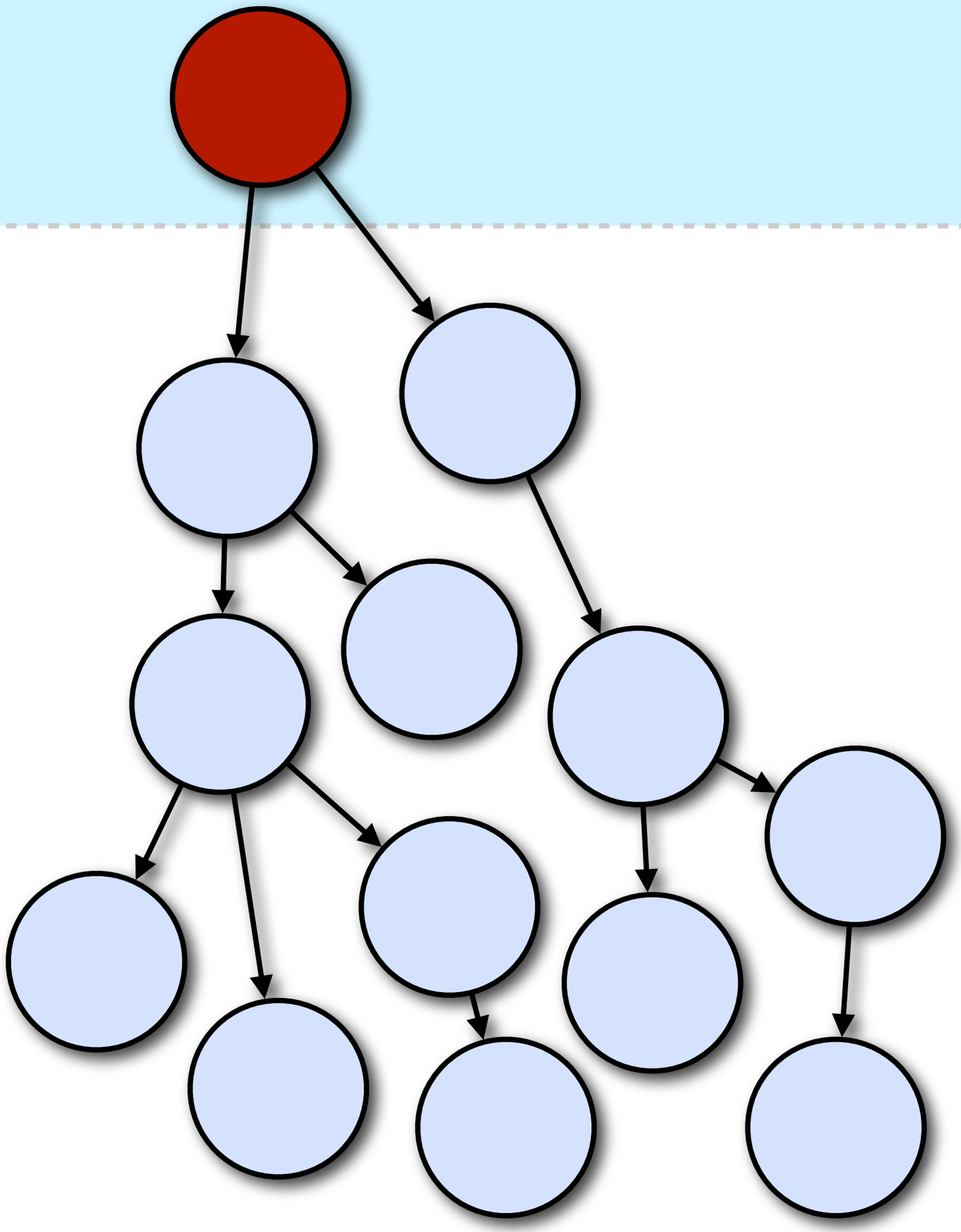

Error
Kernel

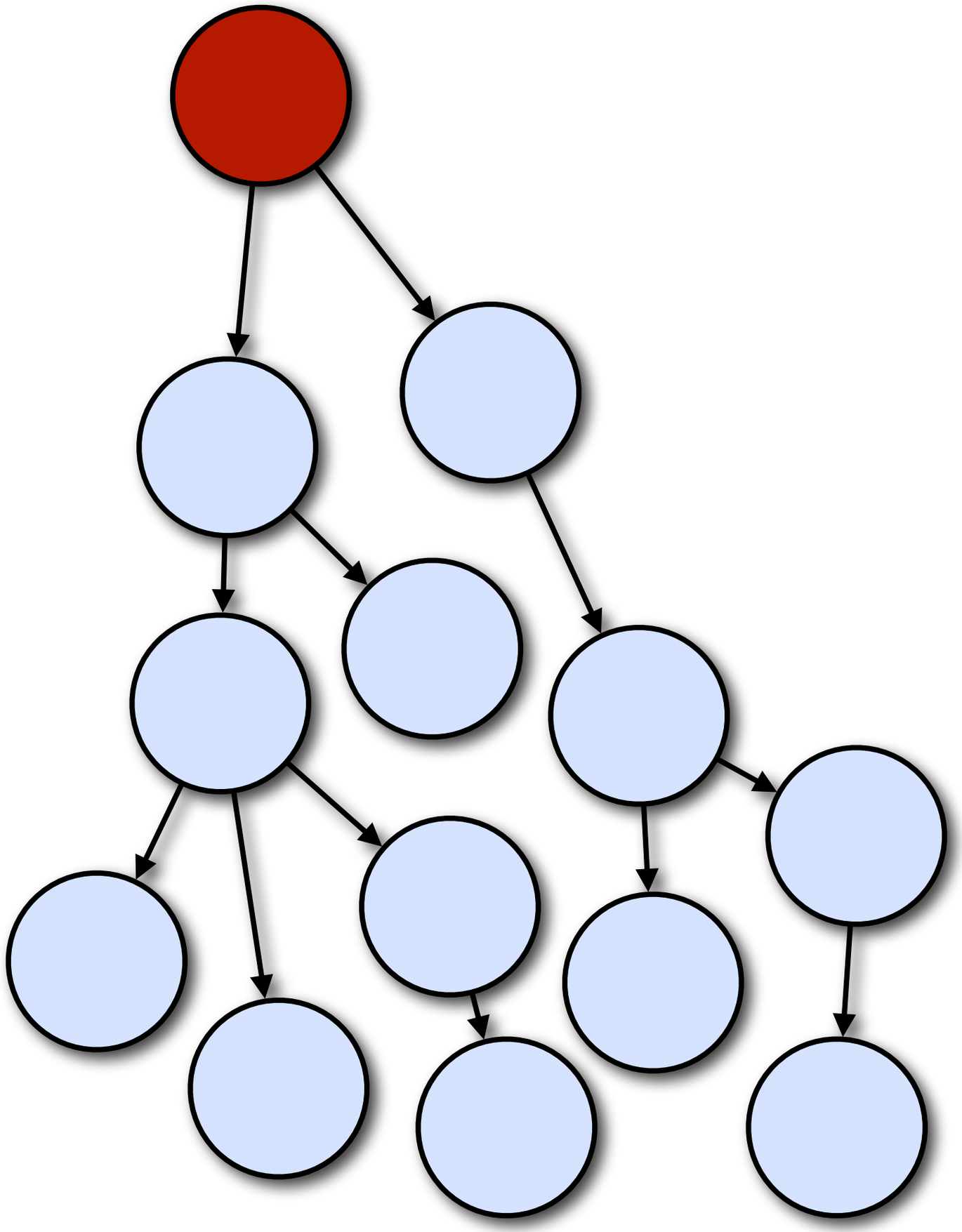

Error
Kernel

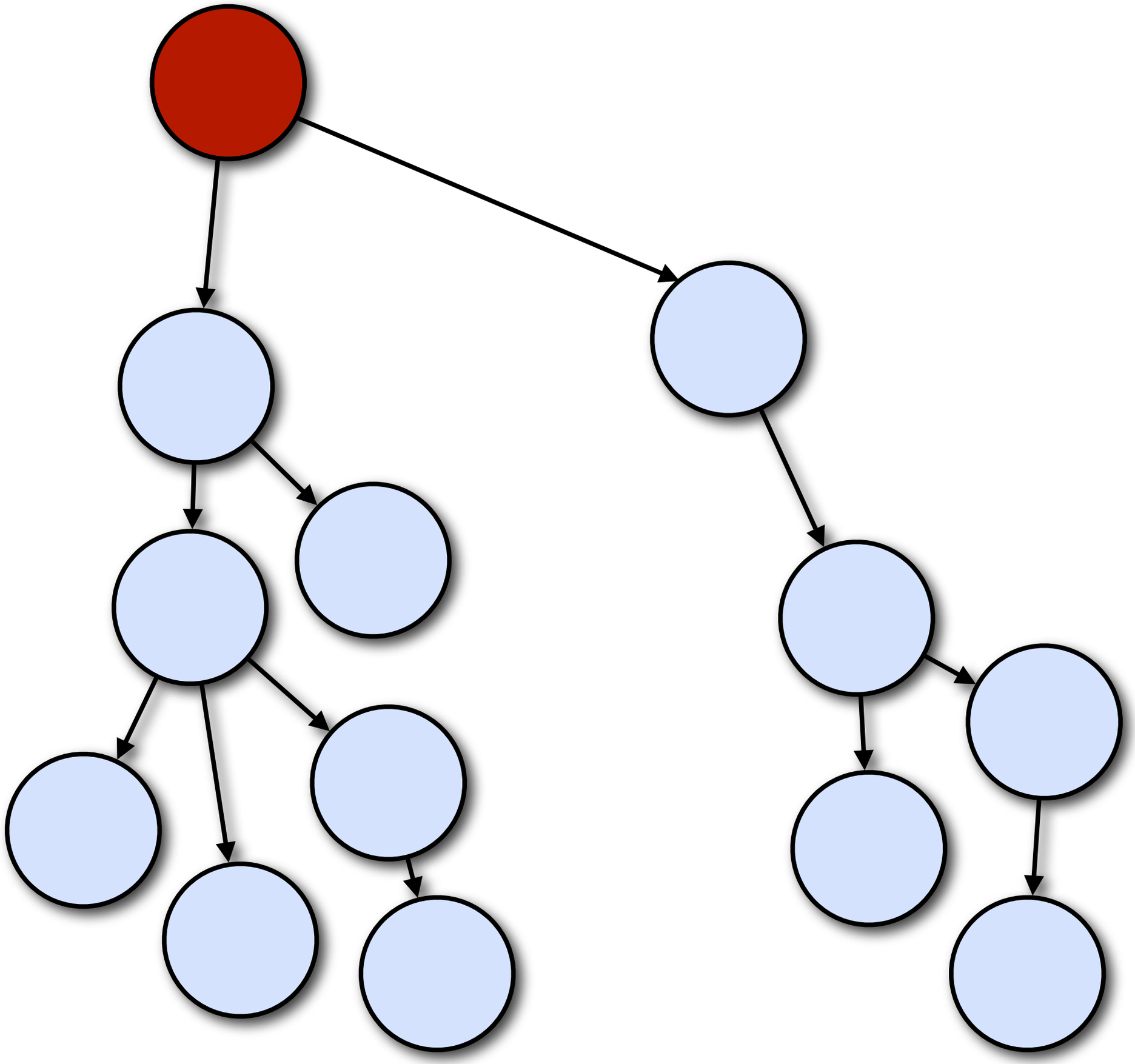

Error
Kernel

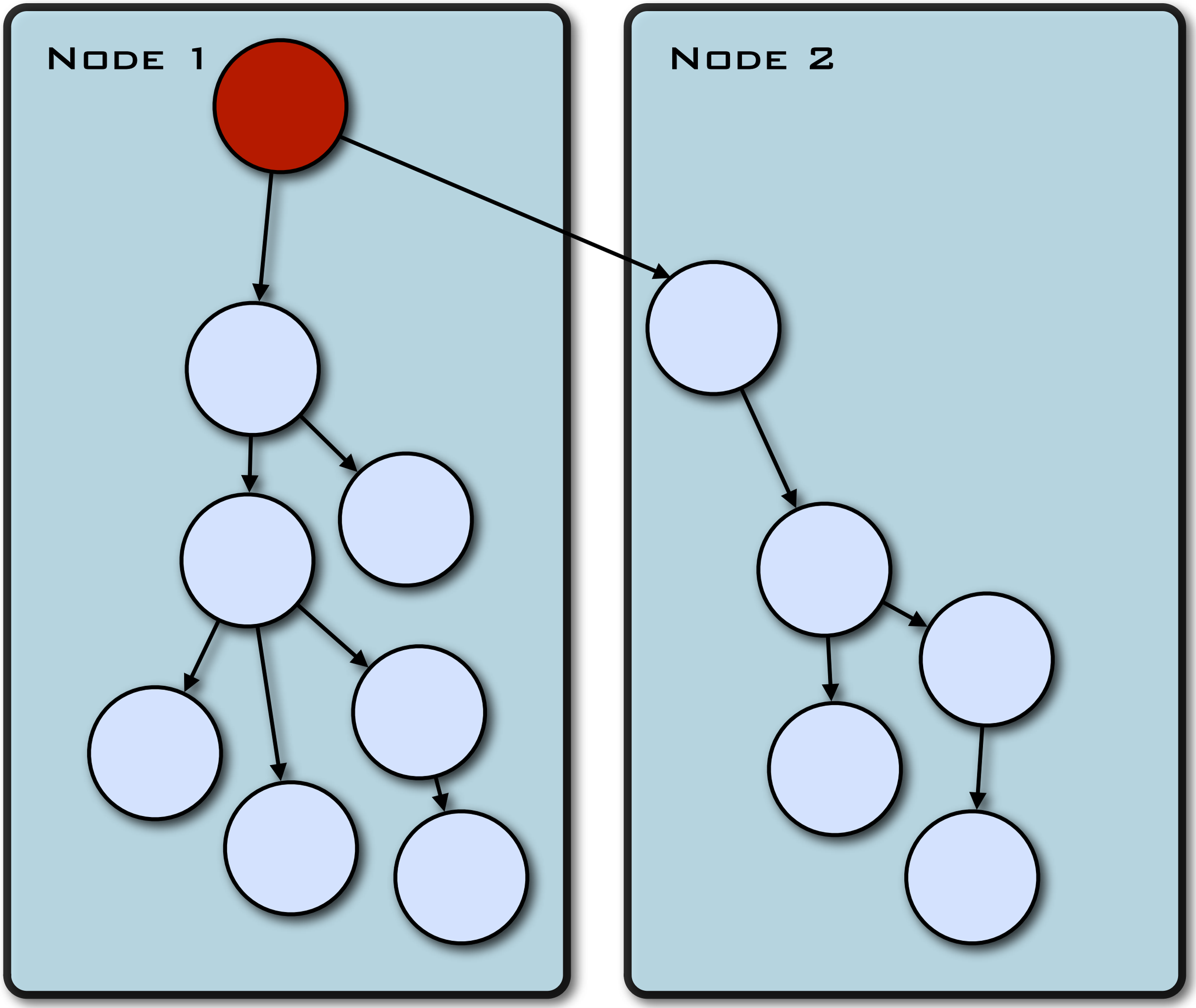

Error
Kernel

Error Kernel








Node 1
Node 2

Linking

```
link(actor)
unlink(actor)

startLink(actor)
spawnLink[MyActor]
```

Fault handlers

```
AllForOneStrategy(
   errors,
   maxNrOfRetries,
   withinTimeRange)

OneForOneStrategy(
   errors,
   maxNrOfRetries,
   withinTimeRange)
```

Supervision

```
class Supervisor extends Actor {
  faultHandler = AllForOneStrategy(
    List(classOf[IllegalStateException])
    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
  }
}
```

...and much much more

STM

FSM

HTTP

Camel

Microkernel

Guice

JTA

Dataflow

OSGi

AMQP

scalaz

Spring

Security

Akka 1.1

To be released today

Get it and learn more

http://akka.io

EOF