Functional Systems

Or: *Functional* Functional Programming

Marius Eriksen • Twitter Inc.
@marius • QCon San Francisco ‘14

Caveat emptor

Where am I coming from?

- 1k+ engineers working on
- a large scale Internet service
- I build systems — I’m not a PL person

I’m not attempting to be unbiased — this is part experience report.

Systems

Systems design is largely about managing complexity.

Need to reduce *incidental* complexity as much as possible.

We'll explore the extent languages help here.

The language isn’t the whole story

@marius
@marius

Follow

Don’t put so much faith in the ability of any programming language to solve all your software engineering problems.

9:01 PM - 1 Jul 2014

37 RETWEETS **42** FAVORITES

Three pieces

Three specific ways in which we've used functional programming for great profit in our systems work:

1. Your server as a function
2. Your state machine as a formula
3. Your software stack as a value

1.

Your server as a function

Modern server software

Highly concurrent

Part of larger distributed systems

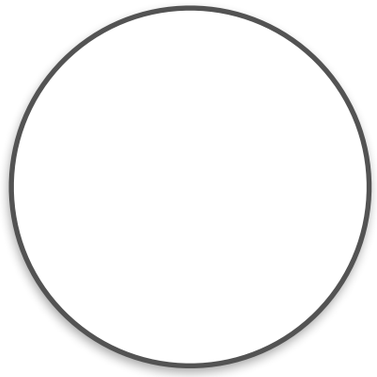
Complicated operating environment

- Asynchronous networks
- Partial failures
- Unreliable machines

Need to support many protocols

Futures

Futures are containers for value

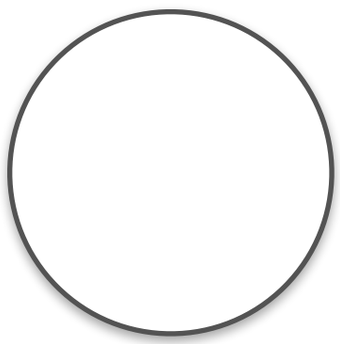


Pending

Successful

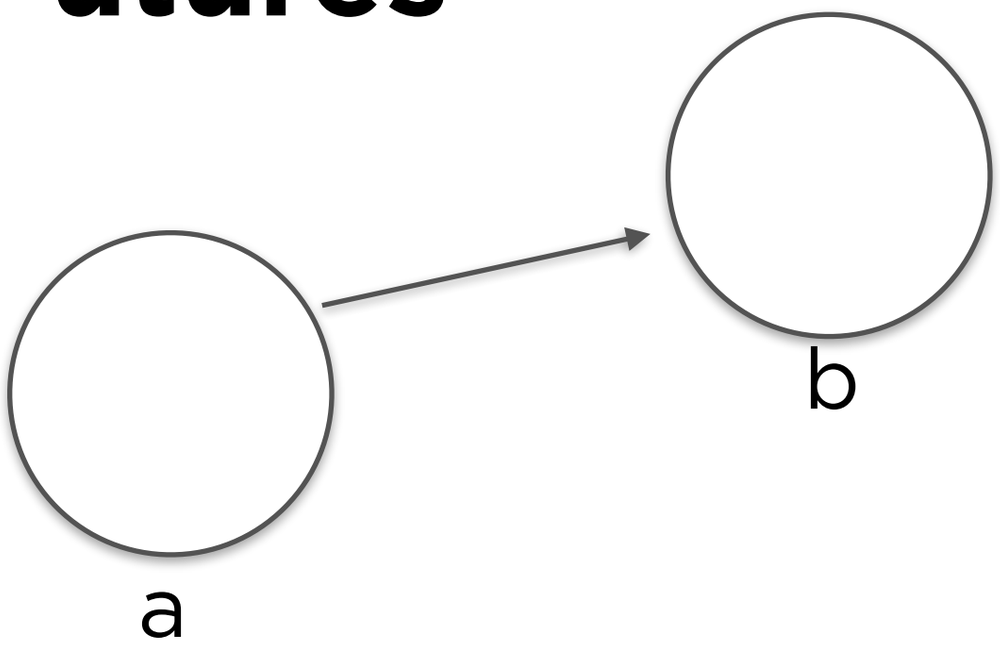
Failed

Futures



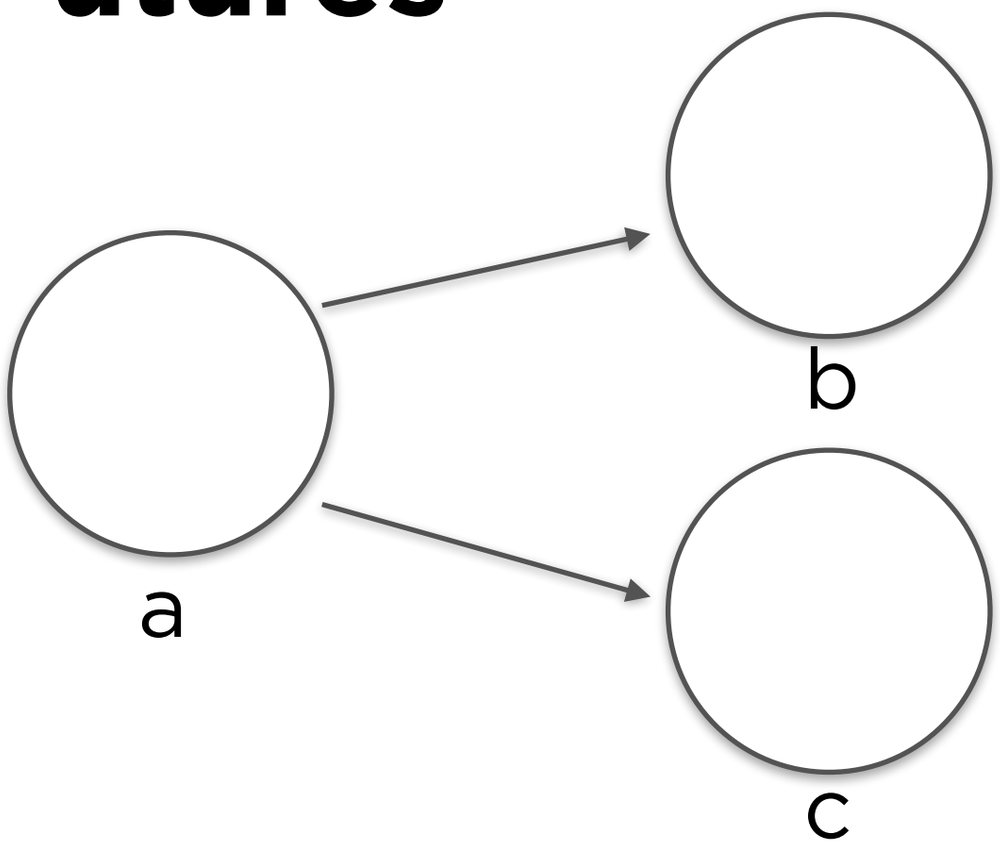

```
val a: Future[Int]
```

Futures



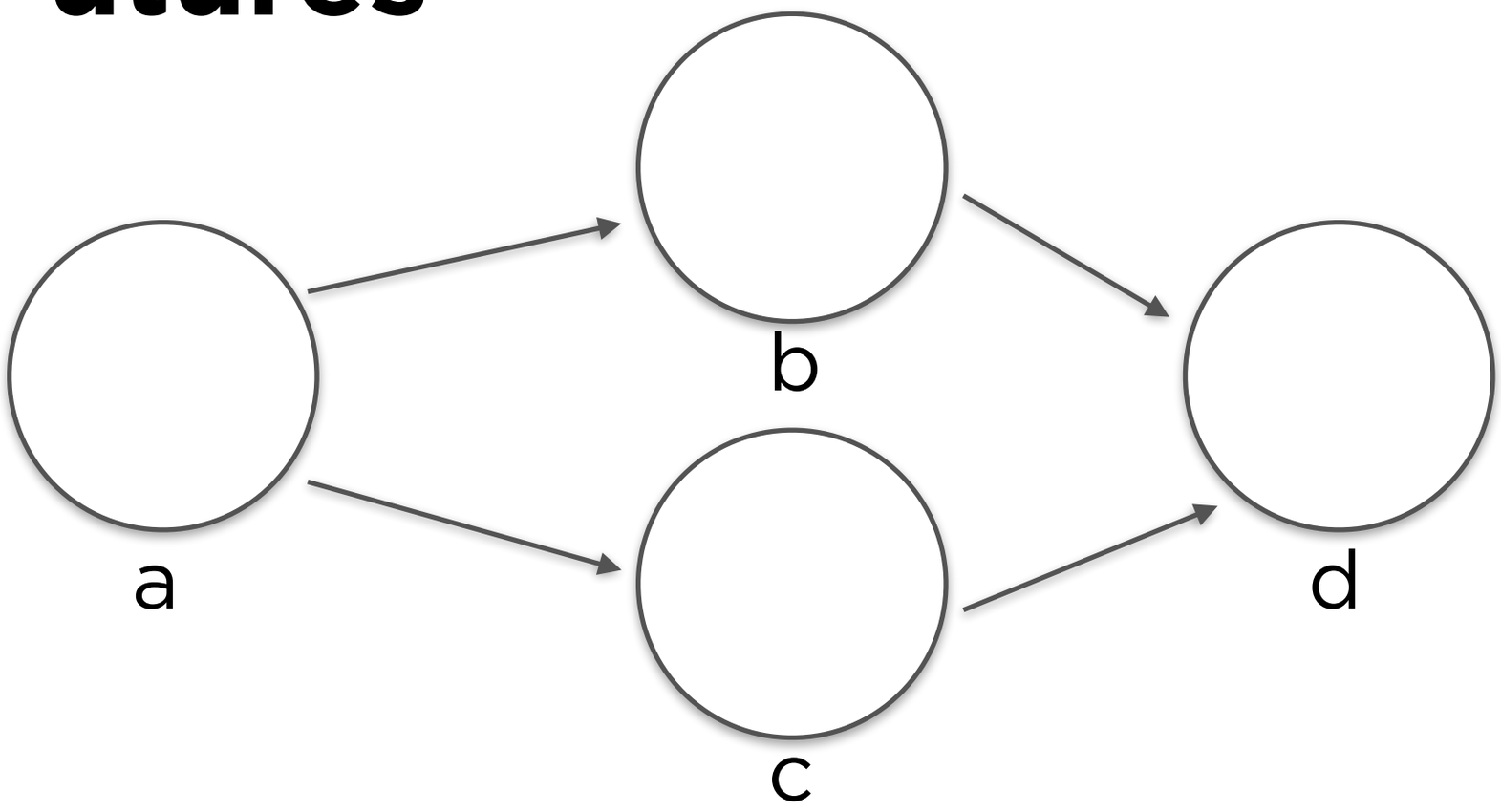

```
val a: Future[Int]
val b = a map { x => x + 512 }
```

Futures



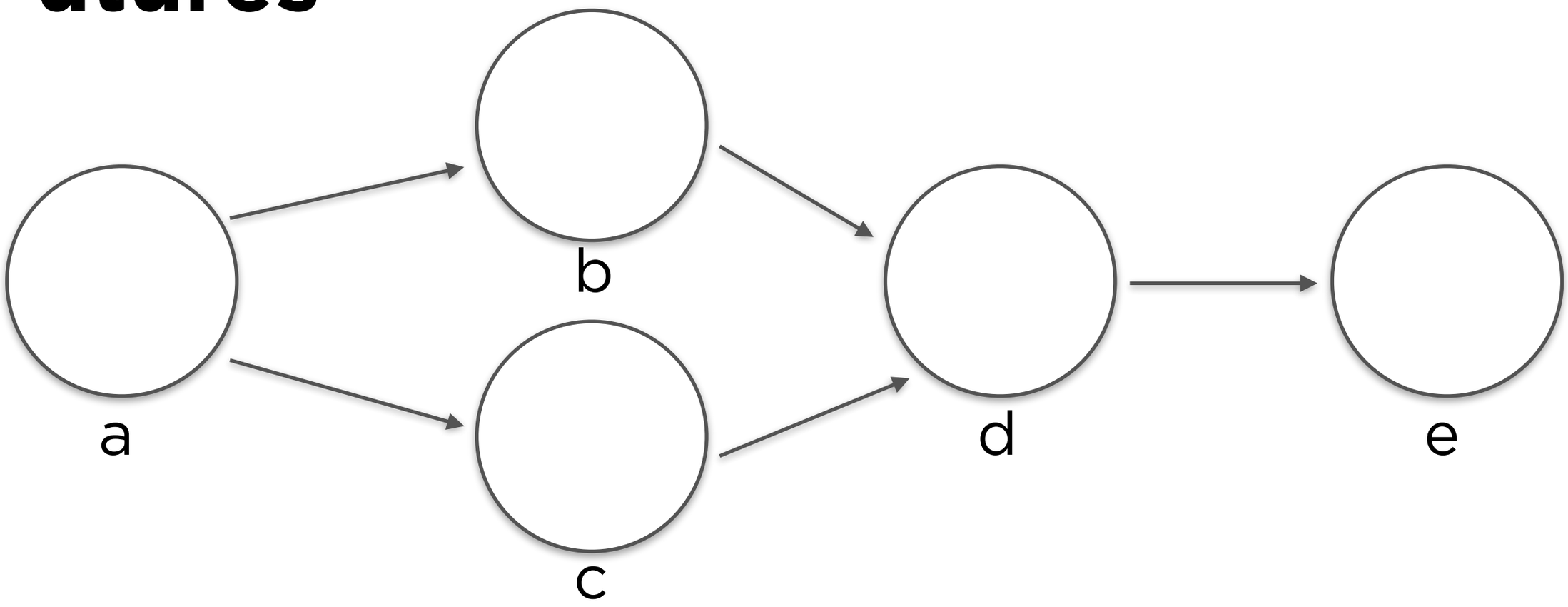

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
```

Futures



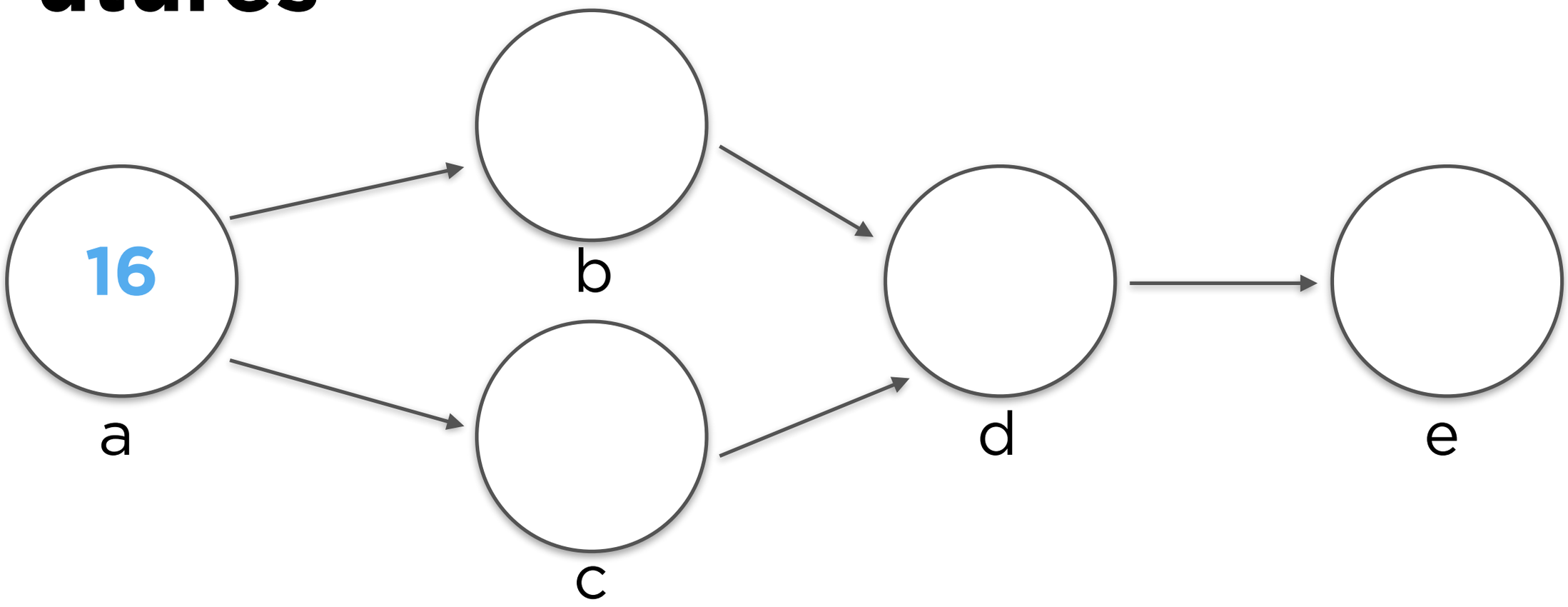

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
```

Futures



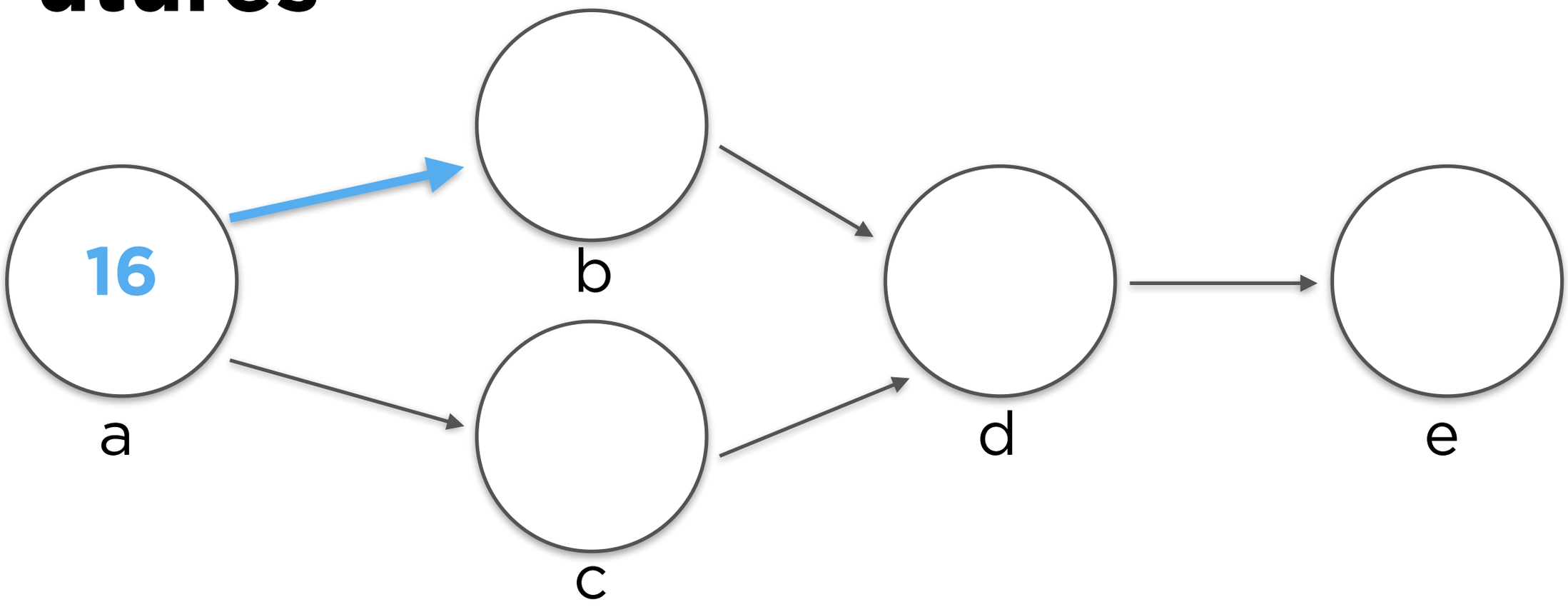

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



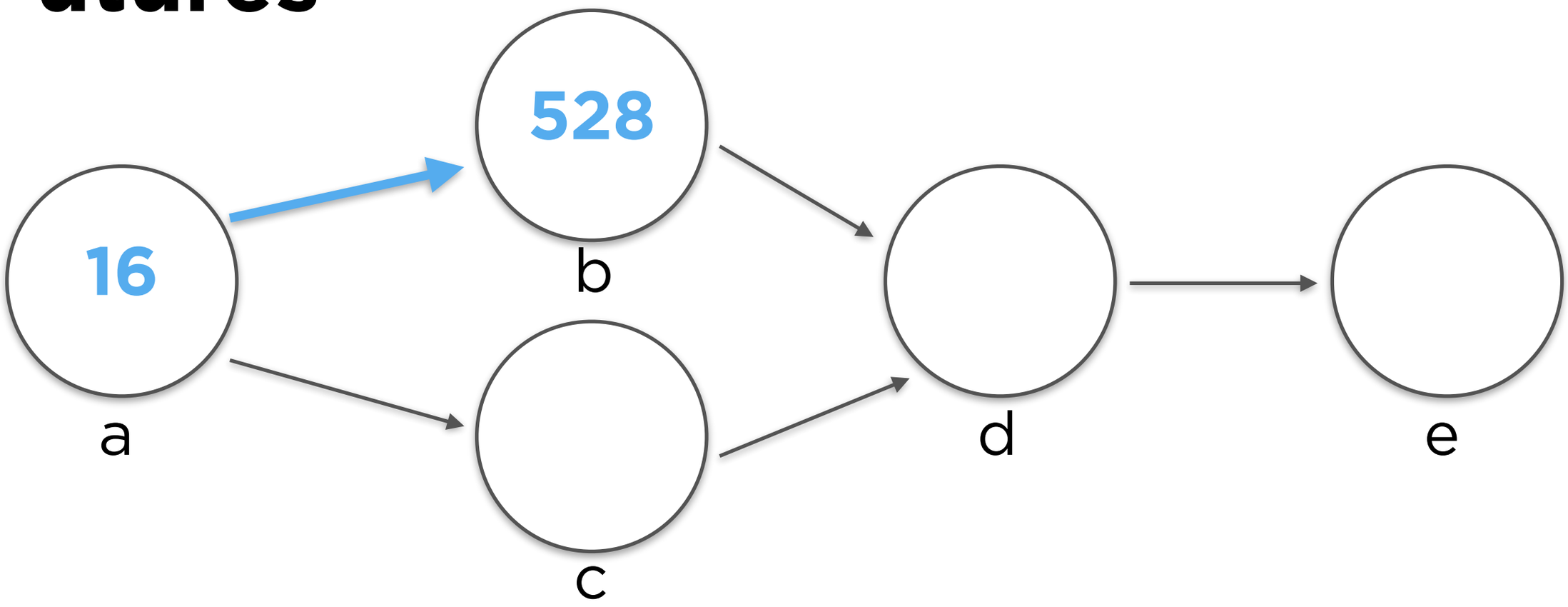

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



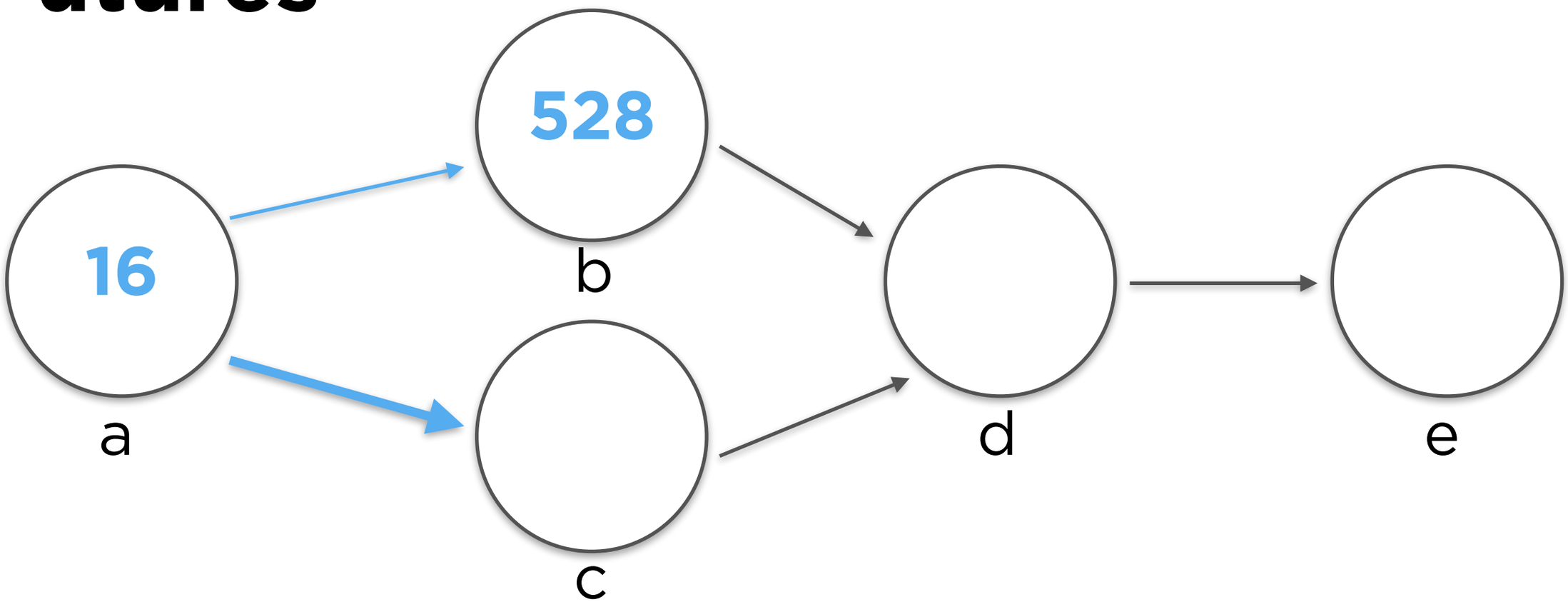

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



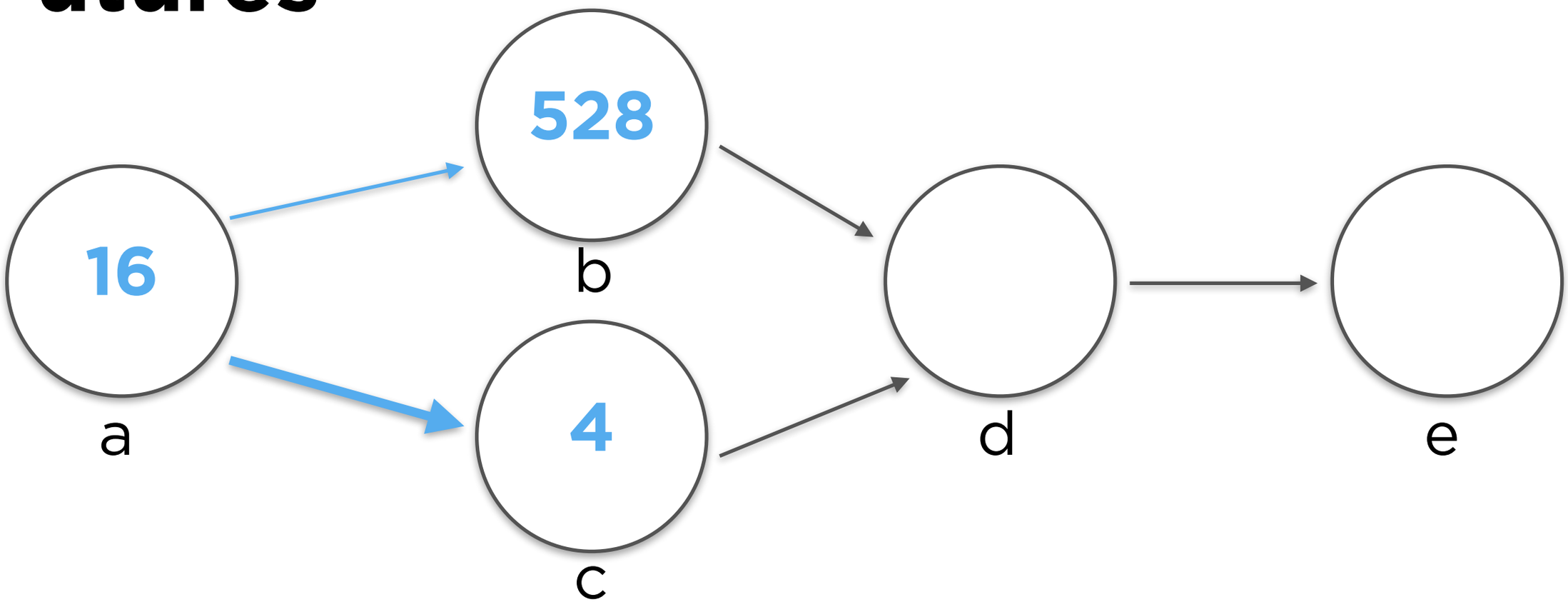

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



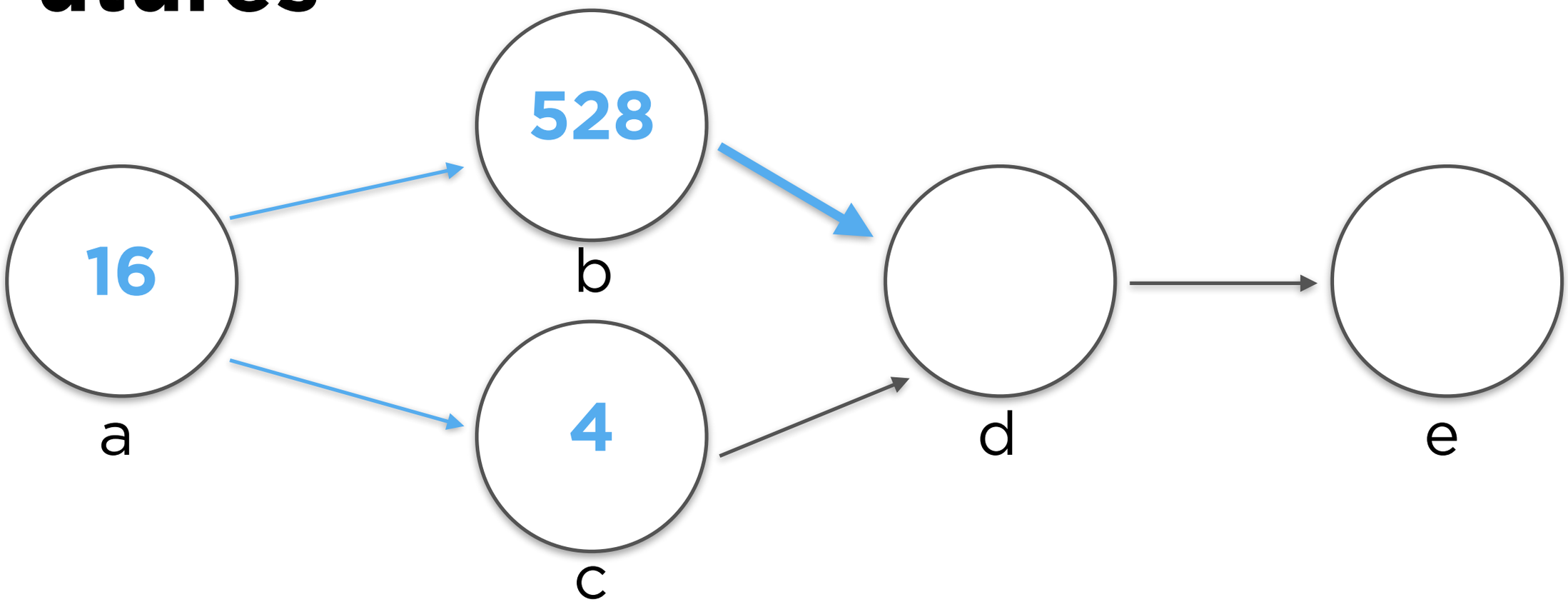

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



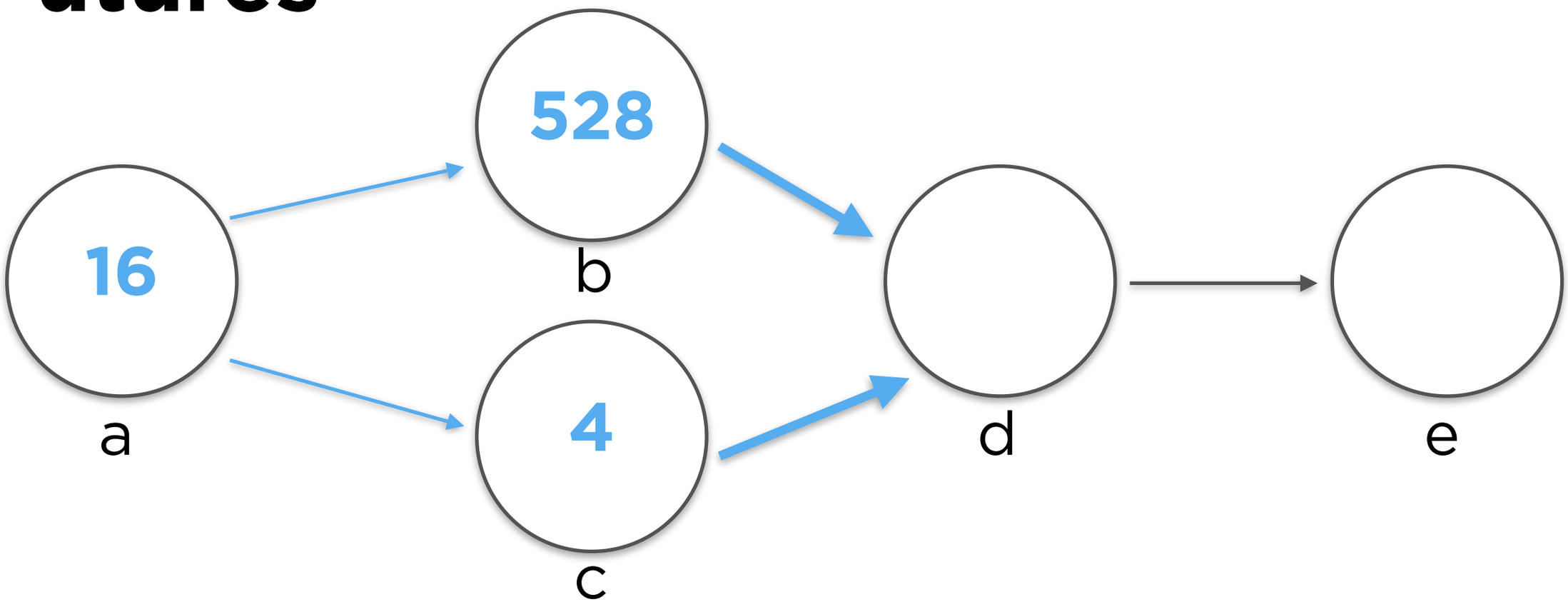

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



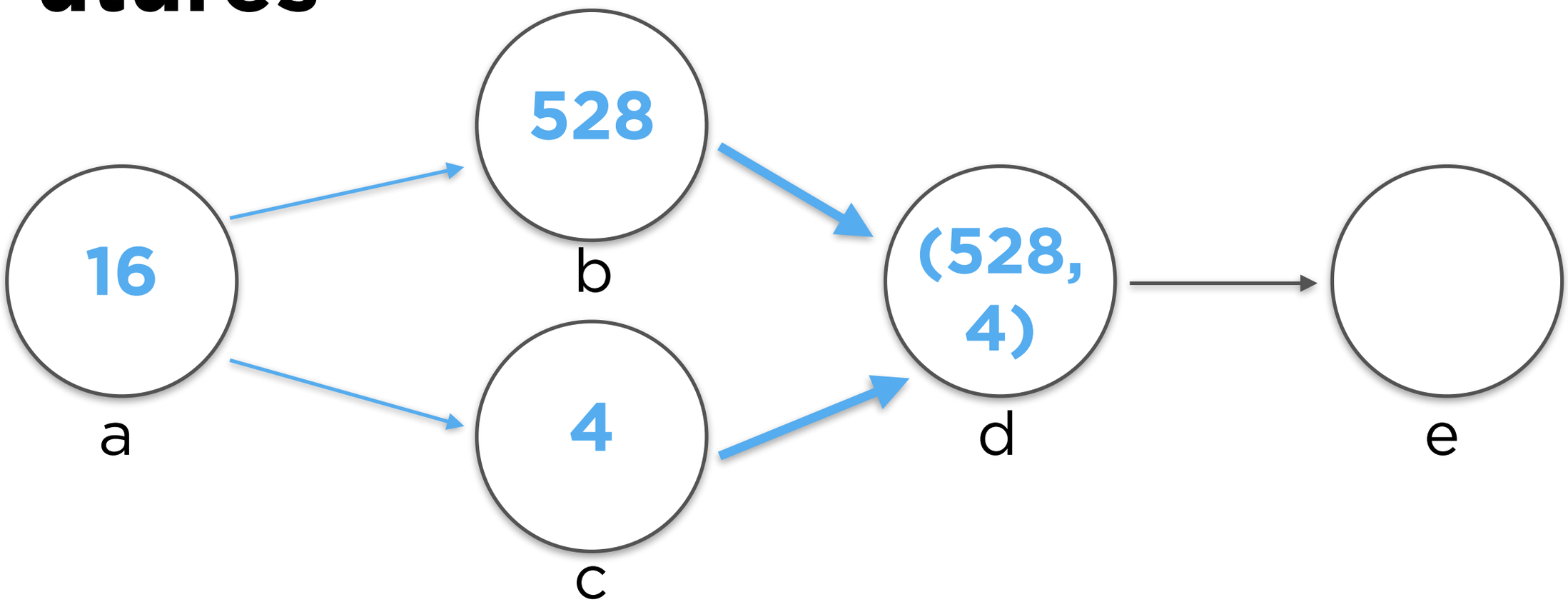

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



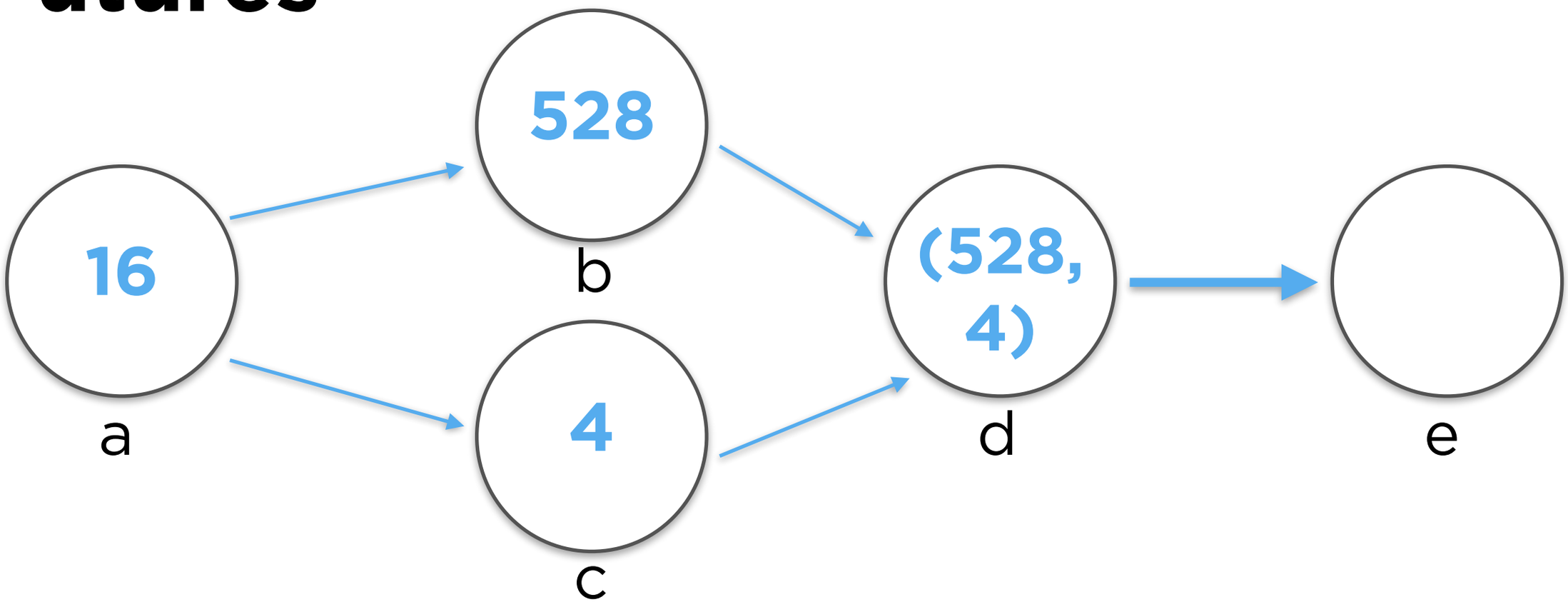

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



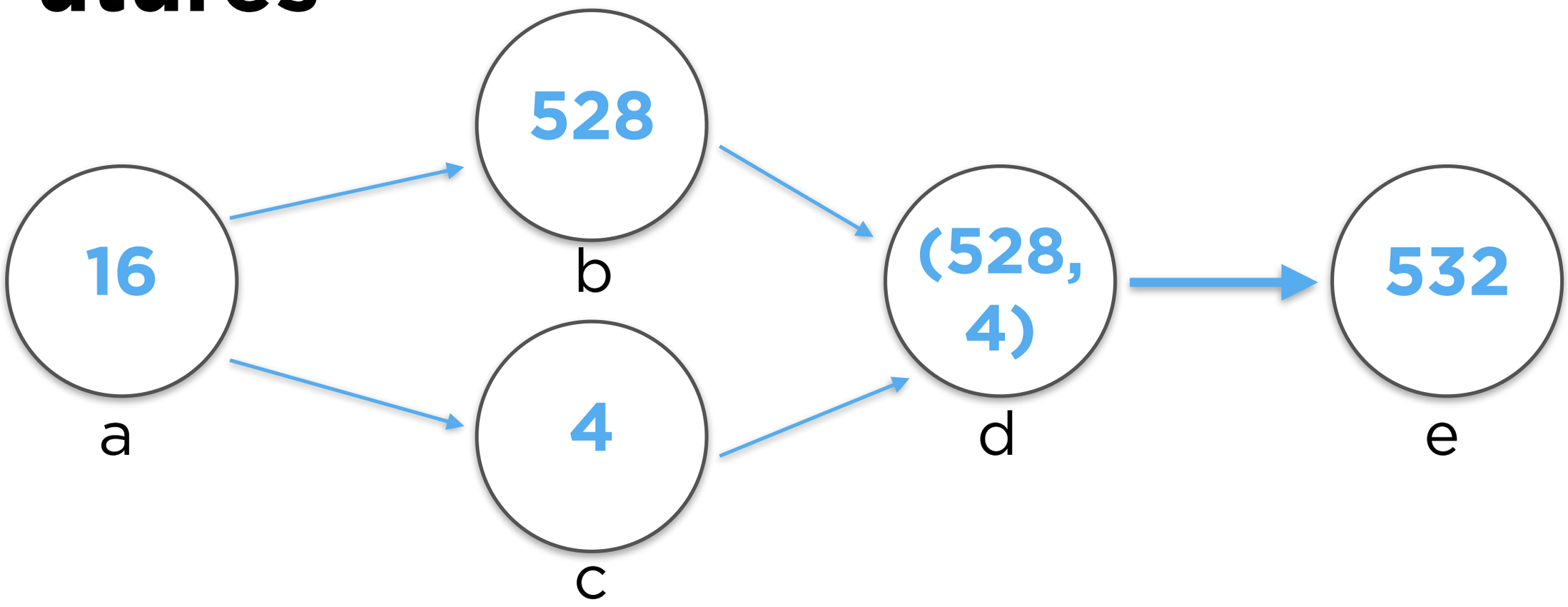

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



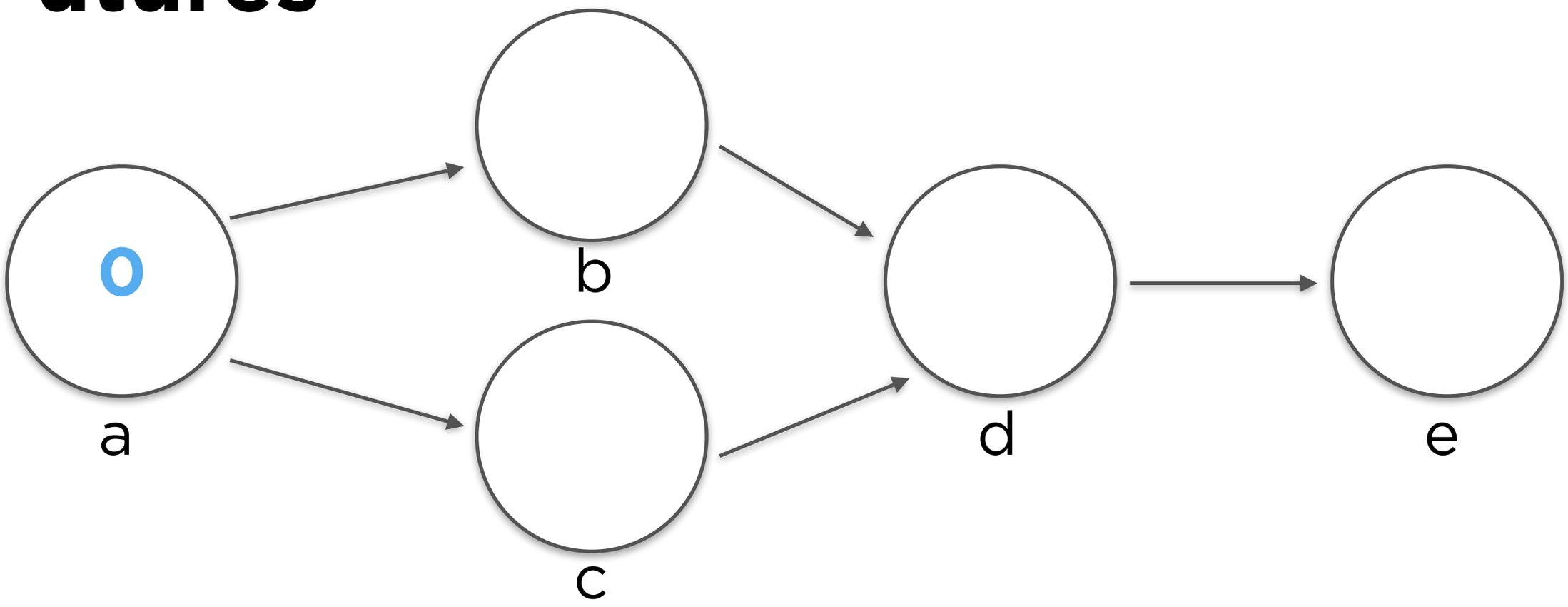

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



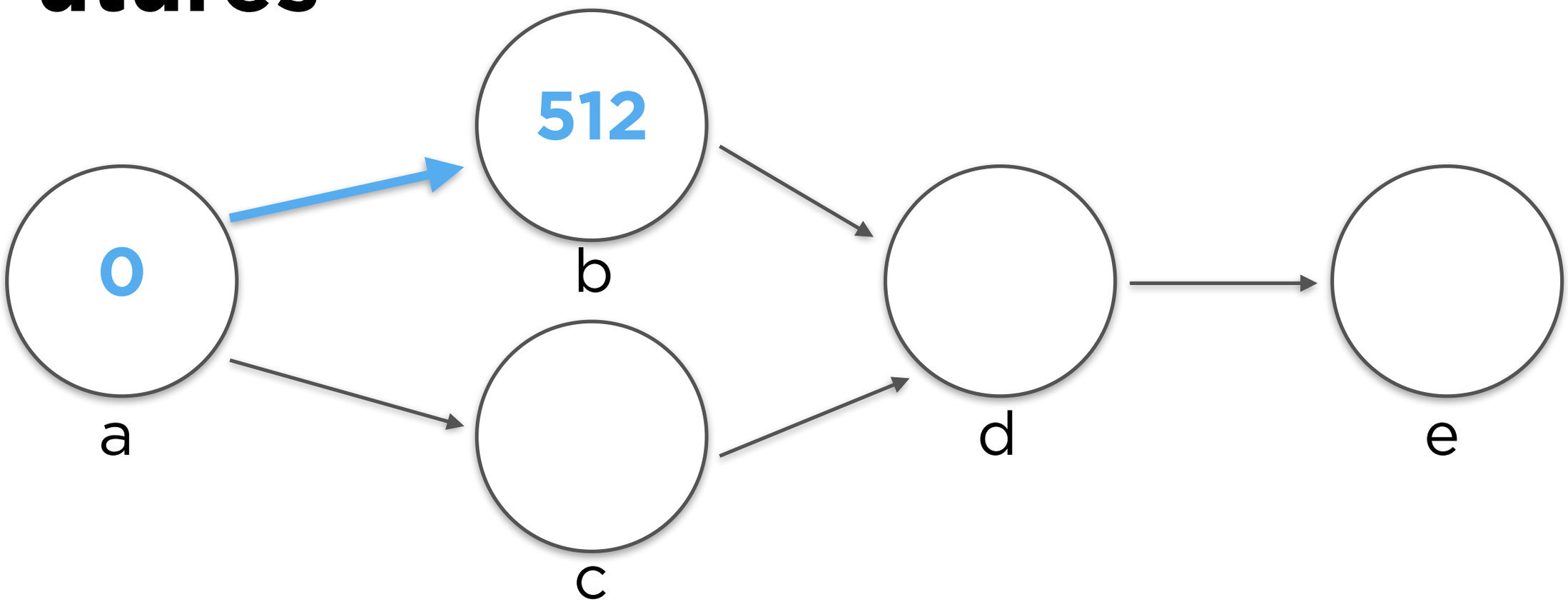

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



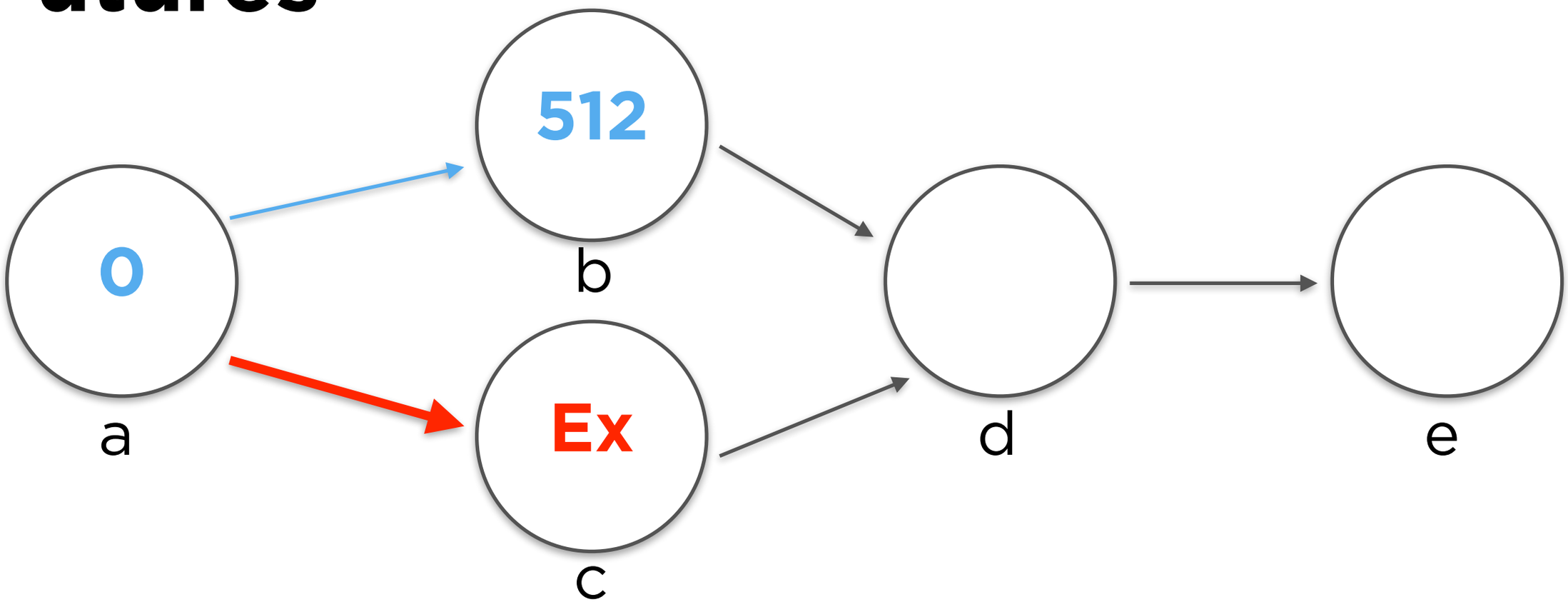

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



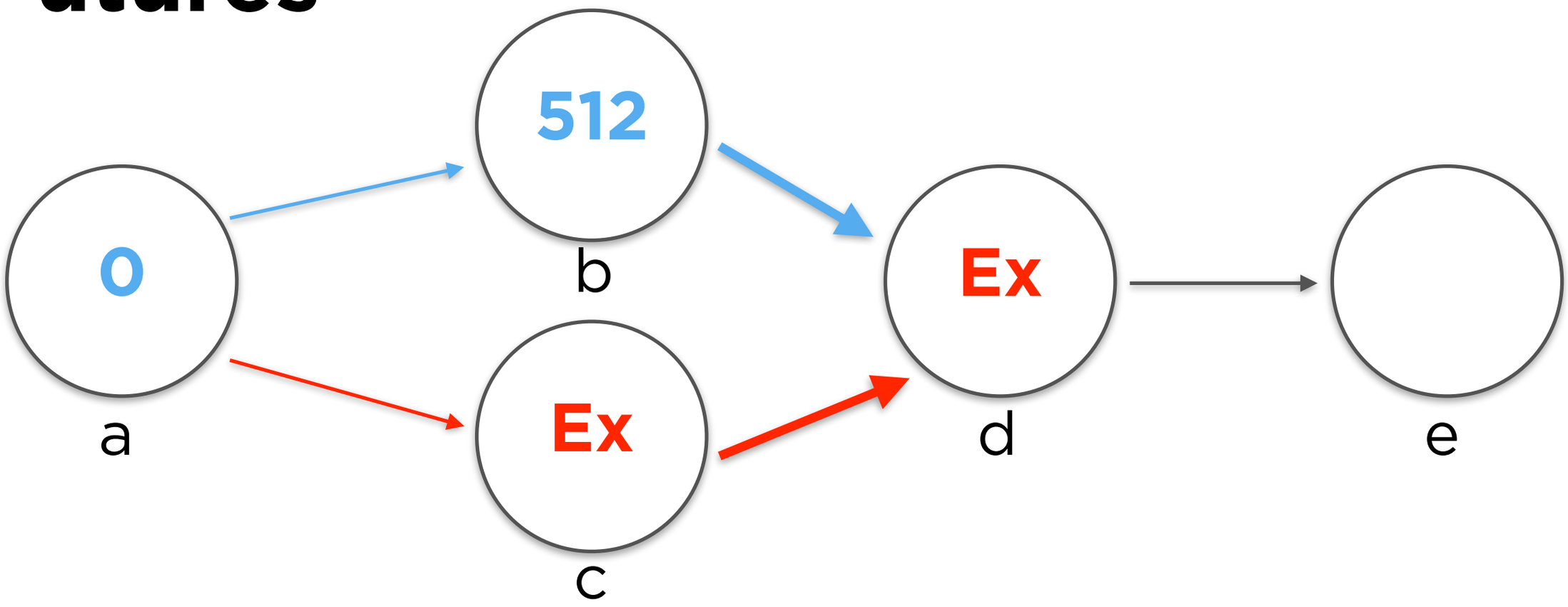

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures



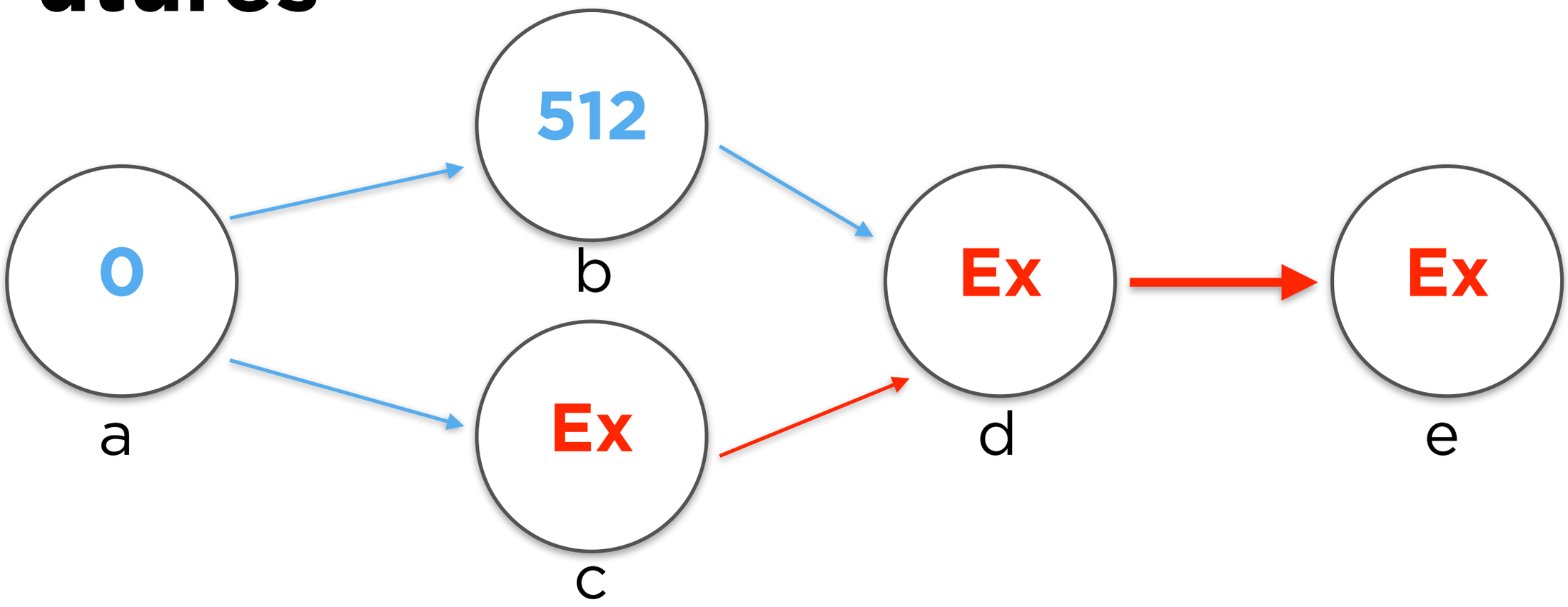

```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures




```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Futures




```
val a: Future[Int]
val b = a map { x => x + 512 }
val c = a map { x => 64 / x }
val d = Future.join(b,c)
val e = d map { case (x, y) => x + y }
```

Dependent composition

Futures may also be defined as a function of other futures. We call this dependent composition.

```
Future[T].flatMap[U](
  f: T => Future[U]): Future[U]
```

Given a `Future[T]`, produce a new `Future[U]`.
The returned `Future[U]` behaves as `f` applied to `t`.

The returned `Future[U]` fails if the outer `Future[T]` fails.

Flatmap

```
def auth(id: Int, pw: String): Future[User]
def get(u: User): Future[UserData]
```

```
def getAndAuth(id: Int, pw: String)
    : Future[UserData]
  = auth(id, pw) flatMap { u => get(u) }
```

Composing errors

Futures recover from errors by another form of dependent composition.

```
Future[T].rescue(
  PartialFunction[Throwable,Future[T]])
```

Like `flatMap`, but operates over exceptional futures.

Rescue

```
val f = auth(id, pw) rescue {
  case Timeout => auth(id, pw)
}
```

(This amounts to a single retry.)

Multiple dependent composition

```
Future.collect[T](fs: Seq[Future[T]])
  : Future[Seq[T]]
```

Waits for all futures to succeed, returning the sequence of returned values.

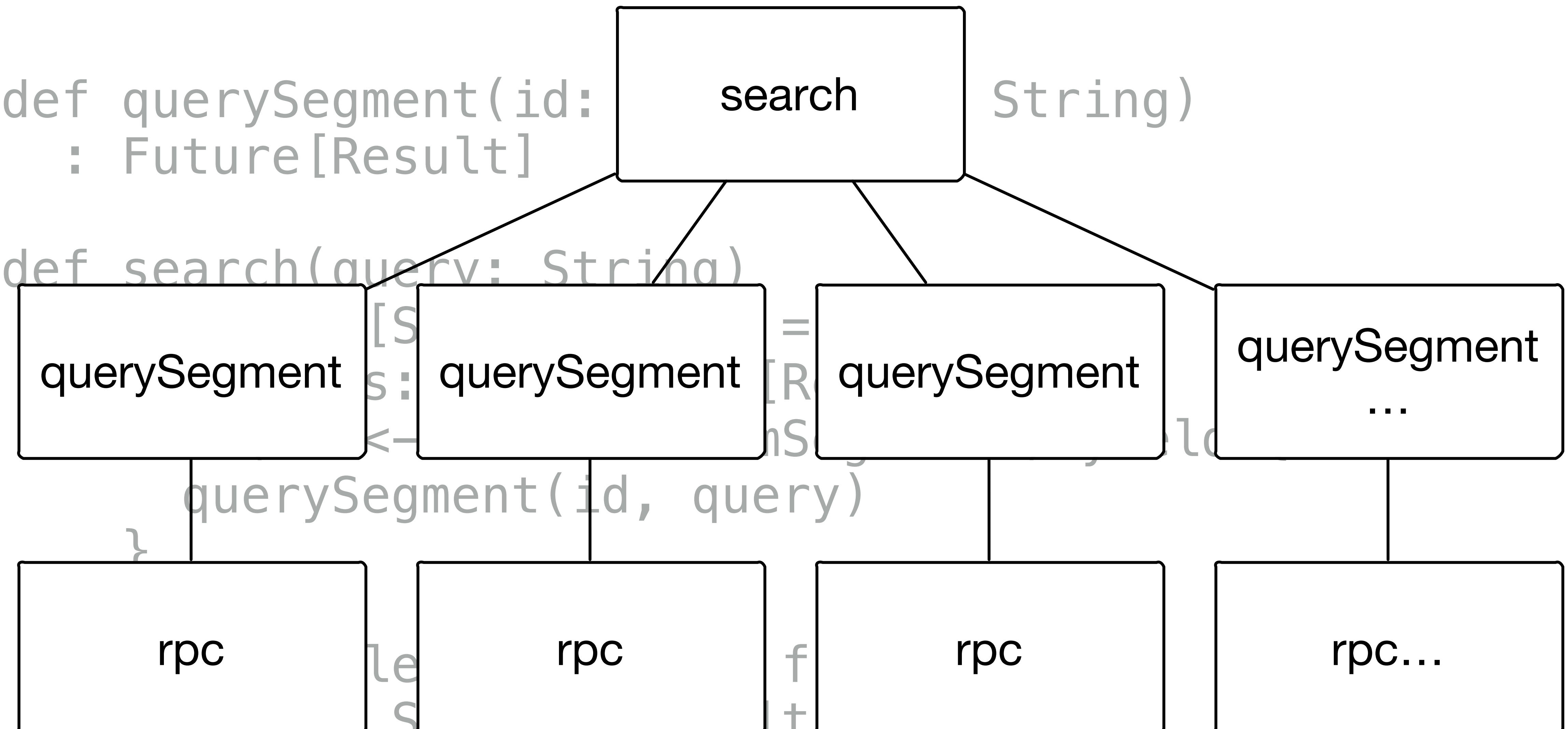
The returned future fails should any constituent future fail.

Segmented search

```
def querySegment(id: Int, query: String)
  : Future[Result]

def search(query: String)
    : Future[Set[Result]] = {

  val queries: Seq[Future[Result]] =
    for (id <- 0 until NumSegments) yield {
      querySegment(id, query)
    }

  Future.collect(queries) flatMap {
    results: Seq[Set[Result]] =>
      Future.value(results.flatten.toSet)
  }
}
```

Segmented search

```
def querySegment(id:          String)
  : Future[Result]

def search(query: String)

      querySegment(id, query)
    }

      Future.value(results.flatten.toSet)
  }
}
```



Services

A service is a kind of **asynchronous function**.

```
trait Service[Req, Rep]
    extends (Req => Future[Rep])

val http:   Service[HttpReq, HttpRep]
val redis:  Service[RedisCmd, RedisRep]
val thrift: Service[TFrame, TFrame]
```

Services are symmetric

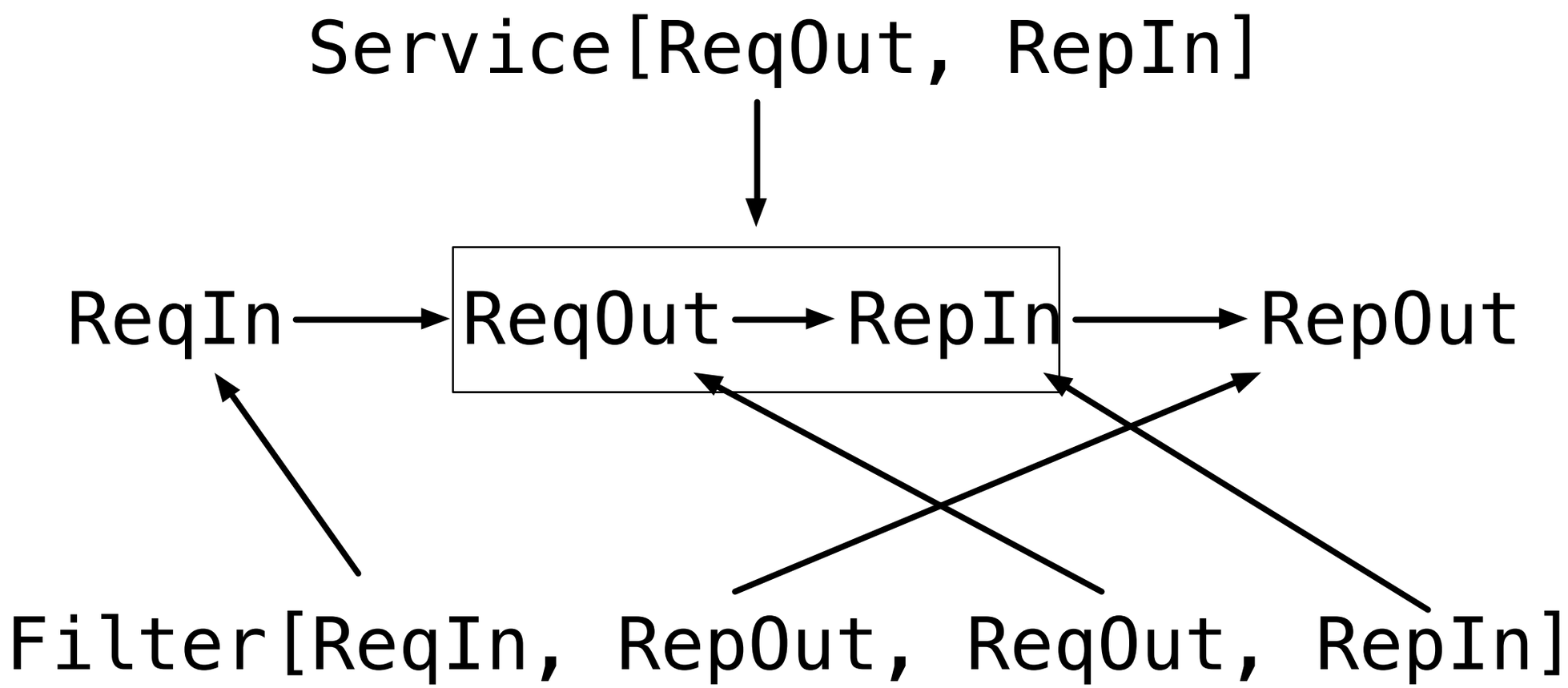
```
// Client:
val http = Http.newService(..)

// Server:
Http.serve(..,
  new Service[HttpReq, HttpRep] {
    def apply(..) = ..
  }
)

// A proxy:
Http.serve(.., Http.newService(..))
```

Filters

Services represent logical endpoints; filters embody **service agnostic behavior** such as:

- Timeouts
- Retries
- Statistics
- Authentication
- Logging

```
trait Filter[ReqIn, ReqOut, RepIn, RepOut]
  extends
((ReqIn, Service[ReqOut, RepIn]) => Future[RepOut])
```



Example: timeout

```
class TimeoutFilter[Req, Rep](to: Duration)
    extends Filter[Req, Rep, Req, Rep] {
  def apply(req: Req, svc: Service[Req, Rep]) =
    svc(req).within(to)
}
```

Example: authentication

```
class AuthFilter extends
 Filter[HttpReq, AuthHttpReq, HttpReq, HttpRep]
{
  def apply(
      req: HttpReq,
      svc: Service[AuthHttpReq, HttpRep]) =
    auth(req) match {
      case Ok(authreq) => svc(authreq)
      case Failed(exc) => Future.exception(exc)
    }
}
```

Combining filters and services

```
val timeout = new TimeoutFilter(1.second)
val auth = new AuthFilter

val authAndTimeout = auth andThen timeout

val service: Service[..] = ..

val authAndTimeoutService =
  authAndTimeout andThen service
```

Real world filters

```
recordHandletime      andThen
traceRequest          andThen
collectJvmStats       andThen
parseRequest          andThen
logRequest            andThen
recordClientStats     andThen
sanitize              andThen
respondToHealthCheck  andThen
applyTrafficControl   andThen
virtualHostServer
```

A. Backup request filter

```
class BackupRequestFilter[Req, Rep](
    quantile: Int,
    range: Duration,
    timer: Timer,
    statsReceiver: StatsReceiver,
    history: Duration,
    stopwatch: Stopwatch = Stopwatch
) extends SimpleFilter[Req, Rep] {
  require(quantile > 0 && quantile < 100)
  require(range < 1.hour)

  private[this] val histo = new LatencyHistogram(range, history)
  private[this] def cutoff() = histo.quantile(quantile)

  private[this] val timeouts = statsReceiver.counter("timeouts")
  private[this] val won = statsReceiver.counter("won")
  private[this] val lost = statsReceiver.counter("lost")
  private[this] val cutoffGauge =
    statsReceiver.addGauge("cutoff_ms") { cutoff().inMilliseconds.toFloat }

  def apply(req: Req, service: Service[Req, Rep]): Future[Rep] = {
    val elapsed = stopwatch.start()
    val howlong = cutoff()
    val backup = if (howlong == Duration.Zero) Future.never else {
      timer.doLater(howlong) {
        timeouts.incr()
        service(req)
      } flatten
    }

    val orig = service(req)

    Future.select(Seq(orig, backup)) flatMap {
      case (Return(res), Seq(other)) =>
        if (other eq orig) lost.incr() else {
          won.incr()
          histo.add(elapsed())
        }

        other.raise(BackupRequestLost)
        Future.value(res)
      case (Throw(_), Seq(other)) => other
    }
  }
}
```

```
  private[this] val cutoffGauge =
    statsReceiver.addGauge("cutoff_ms") { cutoff().inMilliseconds.toFloat

  def apply(req: Req, service: Service[Req, Rep]): Future[Rep] = {
    val elapsed = stopwatch.start()
    val howlong = cutoff()
    val backup = if (howlong == Duration.Zero) Future.never else {
      timer.doLater(howlong) {
        timeouts.incr()
        service(req)
      } flatten
    }

    val orig = service(req)

    Future.select(Seq(orig, backup)) flatMap {
      case (Return(res), Seq(other)) =>
        if (other eq orig) lost.incr() else {
          won.incr()
          histo.add(elapsed())
        }

        other.raise(BackupRequestLost)
        Future.value(res)
      case (Throw(_), Seq(other)) => other
    }
  }
}
```

Futures, services, & filters

In combination, these form a sort of orthogonal basis on which we build our server software.

The style of programming encourages good modularity, separation of concerns.

Most of our systems are phrased as big future transformers.

Issues

There are some practical shortcomings in treating futures as persistent values:

1. Decoupling producer from consumer is not always desirable: we often want to cancel ongoing work.
2. It's useful for computations to carry a *context* so that implicit computation state needn't be passed through everywhere.

Interrupts

```
val p = new Promise[Int]
p.setInterruptHandler {
  case Cancelled =>
    if (p.updateIfEmpty(Throw(..)))
      cancelUnderlyingOp()
}

val f = p flatMap …

f.raise(Cancelled)
```

Locals

```
// Locals are equivalent to
// thread-locals, but with arbitrary
// delimitation.

val f, g: Future[Int]
val l = new Local[Int]
l() = 123
f flatMap { i =>
  l() += i
  g map { j =>
    l() + j
  }
}
```

monkey.org/~marius/funsrv.pdf

Your Server as a Function

Marius Eriksen

Twitter Inc.

marius@twitter.com

n a large-scale setting, where systems ex-
urrency and environmental variability, is
the most experienced programmer. Ef-
tness are paramount—goals which have
h modularity, reusability, and flexibility.

Services Systems boundaries are repr
functions called *services*. They prov
form API: the same abstraction re
servers.

Filters Application-agnostic concerns (
thentication) are encapsulated by *filte*

2.

Your state machine as a formula

Service discovery

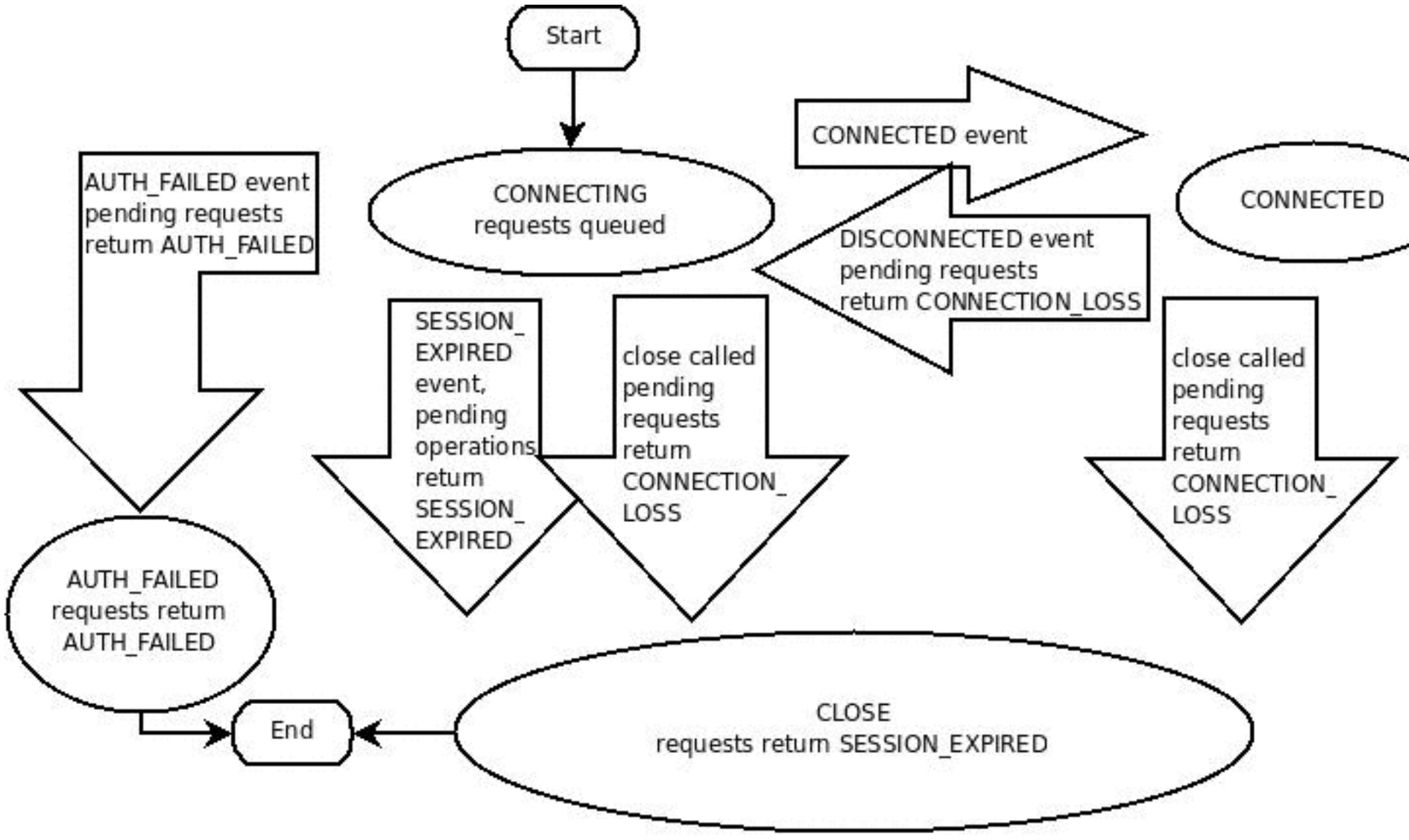
Backed by ZooKeeper

Maintain *convergent view* of the cluster of machines

ZooKeeper is notoriously difficult to deal with correctly

Difficult to reason about the state of your view

In addition, we have to do resource management


Start
CONNECTING
requests queued
AUTH_FAILED event
pending requests
return AUTH_FAILED
SESSION_
EXPIRED
event,
pending
operations
return
SESSION_
EXPIRED
close called
pending
requests
return
CONNECTION_
LOSS
CONNECTED event
DISCONNECTED event
pending requests
return CONNECTION_LOSS
CONNECTED
close called
pending
requests
return
CONNECTION_
LOSS
AUTH_FAILED
requests return
AUTH_FAILED
End
CLOSE
requests return SESSION_EXPIRED

com.twitter.util.Var

```
trait Var[+T] {
  def flatMap[U](f: T => Var[U])
    : Var[U]
  def changes: Event[T]
  …
}

trait Event[+T] {
  def register(s: Witness[T]): Closable
  …
}
```

A simple example

```
val x = Var[Int](2)
val y = Var[Int](1)

val z: Var[Int] = for {
  x0 <- x
  y0 <- y
} yield x0 + y0



// z() == 3
x() = 100  // z() == 101
y() = 100  // z() == 200
```

com.twitter.util.Activity

```
sealed trait State[+T]
case class Ok[T](t: T) extends State[T]
object Pending extends State[Nothing]
case class Failed(exc: Throwable)
    extends State[Nothing]
```

```
case class Activity[+T](
    run: Var[Activity.State[T]]) {
  def flatMap[U](f: T => Activity[U])
    : Activity[U] = …
  …
}
```

Future : val :: Activity : var

A simple wrapper

```
// Turn ZooKeeper operations into
// activities.
case class Zk(underlying: ZooKeeper) {
  def globOf(pat: String)
    : Activity[Seq[String]] = …

  def immutableDataOf(path: String)
    : Activity[Option[Buf]] = …

  def collectImmutableDataOf(paths: Seq[String])
    : Activity[Seq[(String, Option[Buf])]] = {
    def get(path: String)
      : Activity[(String, Option[Buf])] =
      immutableDataOf(path).map(path -> _)
    Activity.collect(paths map get)
  }
}
```

Implementing serversets

```
case class Serverset(zk: Zk) {
  def dataOf(pat: String)
    : Activity[Seq[(String, Option[Buf])]] =
    zk.globOf(pat).flatMap(
      zk.collectImmutableDataOf)

  def parse(Seq[(String, Option[Buf])])
    : Set[SocketAddress] = ..

  def entriesOf(pat: String)
    : Activity[Set[SocketAddress]] =
    dataOf(pat).map(parse)
}
```

Broken ZK clients

```
class VarServerSet(v: Var[ServerSet]) {
  def entriesOf(path: String)
      : Activity[Set[Entry]] = Activity(
      v.flatMap(ss =>
        ss.entriesOf(path.run))
}
```

Retrying ZK instance

```
object Zk {
  // Constructor for dynamic ZK
  // instance.
  def retrying(backoff: Duration)
    : Var[Zk]
  …
}
```

Gluing it together

```
val serverset = VarServerSet(
  Zk.retrying(10.seconds).map(zk =>
    new ServerSet(zk)))

val set =
  serverset.entriesOf(“/foo/bar”)

set.changes.observe({ addrs =>
  updateLoadBalancer(addrs)
})
```

Resource management

```
trait Event[+T] {
  def register(s: Witness[T]): Closable
  …
}
```

```
trait Closable {
  def close(deadline: Time)
    : Future[Unit]
  …
}
```

Composable closable

```
object Closable {
  def all(closables: Closable*)
    : Closable

  def sequence(closables: Closable*)
    : Closable

  val nop: Closable

  def closeOnCollect(
    closable: Closable, obj: Object)

  …
}
```

Resource management

Lifetime of observation is *entirely determined* by consumer. Everything else composes on top.

Anything in the middle (`Var.flatMap`) does not need to be concerned with resource management.

If updates aren’t needed, Vars are closed.

3.

Your software stack as a value

Software configuration

Finagle comprises many modules (filters, services) which compose together to give the emergent behavior we want.

They need to be parameterized:

- systems parameters — e.g. pool sizes, concurrency limits;
- module injection — e.g. stats, logging, tracing.

Prior art: “cake pattern,” dependency injection frameworks (e.g. Guice)

Ours is a more regular world

We can take advantage of the fact that our software is highly compositional: e.g. the entire Finagle stack is expressed in terms of Service composition.

Idea: make it a first class persistent data structure which can be inspected and transformed.

- injecting a parameter is ‘mapping’ over this data structure;
- inserting a module is a transformation

com.twitter.finagle.Stack

```
trait Stack[T] {
  def transform(
    fn: Stack[T] => Stack[T]): Stack[T]

  def ++(right: Stack[T]): Stack[T]
  def +:(stk: Stackable[T]): Stack[T]

  def make(params: Params): T
}
```

Nodes

```
case class Node[T](
    mk: (Params, Stack[T]) => Stack[T],
    next: Stack[T]
  ) extends Stack[T] {
  def make(params: Params) =
    mk(params, next).make(params)
}

case class Leaf[T](t: T)
  extends Stack[T] {
  def make(params: Params) = t
}
```

Parameters

```
// A typeclass for parameter types
trait Param[P] {
  def default: P
}
```

```
trait Params {
  def apply[P: Param]: P
  def contains[P: Param]: Boolean
  def +[P: Param](p: P): Params
}
```

Parameter definition

```
case class Poolsize(min: Int, max: Int)

implicit object Poolsize
  extends Stack.Param[Poolsize] {
  val default =
    Poolsize(0, 100)
}
```

Parameter use

```
val params: Params

val Poolsize(min, max) = params[Poolsize]
…
new Pool(min, max)
```

Modules

```
object FailFast {
  def module[Req, Rep] =
  new Stack.Module2[Stats, Timer, ServiceFactory[Req, Rep]] {
  def make(
    stats: Stats,
    timer: Timer,
    next: ServiceFactory[Req, Rep]) =
  new FailFastFactory(
    next, stats.scope("failfast"), timer)
}
```

Building

```
val stk = new StackBuilder[ServiceFactory[Req, Rep]]
  (nilStack[Req, Rep])

stk.push(ExpiringService.module)
stk.push(FailFastFactory.module)
stk.push(DefaultPool.module)
stk.push(TimeoutFilter.module)
stk.push(FailureAccrualFactory.module)
stk.push(StatsServiceFactory.module)
stk.push(StatsFilter.module)
stk.push(ClientDestTracingFilter.module)
stk.push(MonitorFilter.module)
stk.push(ExceptionSourceFilter.module)

val stack: Stack[ServiceFactory[Req, Rep]] =
  stk.result
```

Using

```
val params = Params.empty +
  Stats(statsReceiver) +
  Poolsize(10, 50) +
  …

val factory: ServiceFactory[..] =
  stack.make(params)
```

Modifying

```
val muxStack = stdStack
  .replace(Pool, ReusingPool.module)
  .replace(PrepConn, Leaser.module)
```

Inspection

```
scala> println(StackClient.newStack[Int, Int])
Node(role = prepfactory, description = PrepFactory)
Node(role = tracer,
  description = Handle span lifecycle events to report tracing from protocols)
Node(role = servicecreationstats,
  description = Track statistics on service creation failures and .. latency)
Node(role = servicetimeout,
  description = Time out service acquisition after a given period)
Node(role = requestdraining, description = RequestDraining)
Node(role = loadbalancer, description = Balance requests across multiple
endpoints)
Node(role = exceptionsource, description = Source exceptions to the service name)
Node(role = monitoring, description = Act as last-resort exception handler)
Node(role = endpointtracing, description = Record remote address of server)
Node(role = requeststats, description = Report request statistics)
Node(role = factorystats, description = Report connection statistics)
Node(role = failureaccrual,
  description = Backoff from hosts that we cannot successfully make requests to)
Node(role = requesttimeout, description = Apply a timeout to requests)
Node(role = pool, description = Control client connection pool)
Node(role = failfast,
  description = Backoff exponentially on connection failure)
Node(role = expiration,
  description = Expire a service after a certain amount of idle time)
Node(role = prepconn, description = PrepConn)
Leaf(role = endpoint, description = endpoint)
```

Dynamic inspection

http

- PrepFactory
- Tracer
- ServiceCreationStats
- ServiceTimeout
- RequestDraining
- LoadBalancer
- ExceptionSource
- Monitoring
- EndpointTracing
- RequestStats
- FactoryStats
- FailureAccrual
- RequestTimeout
- Pool
- FailFast

LoadBalancer

Balance requests across multiple endpoints

Parameter	Value
monitor	NullMonitor
reporter	<function2>
label	http
loadBalancerFactory	DefaultBalancerFactory
statsReceiver	StatsReceiver
va	Bound(Set(twitter.com/199.59.148.82:80, twitter.com/199.59.149.198:80, twitter.com/199.59.150.39:80, twitter.com/199.59.148.10:80))
log	Logger

What have we learned?

On abstraction

Abstraction has gotten a bad name because of `AbstractBeanFactoryImpls`.

Rule of thumb: introduce abstraction when it increases precision, when it serves to clarify.

Often, we can use abstraction to make things more *explicit.*

Avoid needless indirection.

Compose

Composability is one of our greatest assets — combine simple parts into a whole with emergent behavior.

- Easy to reason about, test, constituent parts
- Easy to combine in multiple ways
- Enforces modularity

Find your “orthogonal bases.”

- Find abstractions which combine in non overlapping ways

Decouple, separate concerns

Separate *semantics* from *mechanism*; handle problems separately (and reusably)

- Leads to cleaner, simpler systems
  - Simpler user code — pure application logic
  - Flexibility in implementation

Leads to a “software tools” approach to systems engineering.

Keep it simple

Scala, and FP languages generally, are very powerful. Use the simple features that get you a lot of mileage.

When your platonic ideal API doesn't quite fit, it's okay to dirty it up a little, but be careful.

Be mindful of the tradeoffs of static guarantees with simplicity and understandability — always remember the reader!

Software engineering is in part the art of knowing when to make things *worse*.

Functional programming

Many tools for complexity management: Immutability, rich structures, modularity, strong typing.

It's easier to reason about correctness, but *harder* to reason about performance.

Bridging the platonic world of functional programming to the more practical one requires us to get dirty.

And have fun! Thanks.

@marius

https://finagle.github.io/