Akka:

Simpler Concurrency, Scalability & Fault-tolerance through Actors

Jonas Bonér
jonas@jonasboner.com
twtr: @jboner

We believe that...

- Writing correct concurrent applications is **too hard**

- Scaling out applications is **too hard**

- Writing highly fault-tolerant applications is **too hard**

It doesn't have to be like this

We need to raise the abstraction level

Locks & Threads

are...

...sometimes

plain evil

...but always the

wrong default

"Threads are to Concurrency as Witchcraft is to Physics"

"Hanging by a thread is the punishment for Shared State Concurrency"

- Gilad Bracha

Introducing

STM Actors Agents

Dataflow Distributed Open Source

RESTful Secure Persistent

OVERVIEW

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

Simpler Concurrency

Write simpler concurrent (yet correct) applications using Actors, STM & Transactors.

Event-driven Architecture

The perfect platform for asynchronous event-driven architectures. Never block.

True Scalability

Scale out on multi-core or multiple nodes using asynchronous message passing.

Fault-tolerance

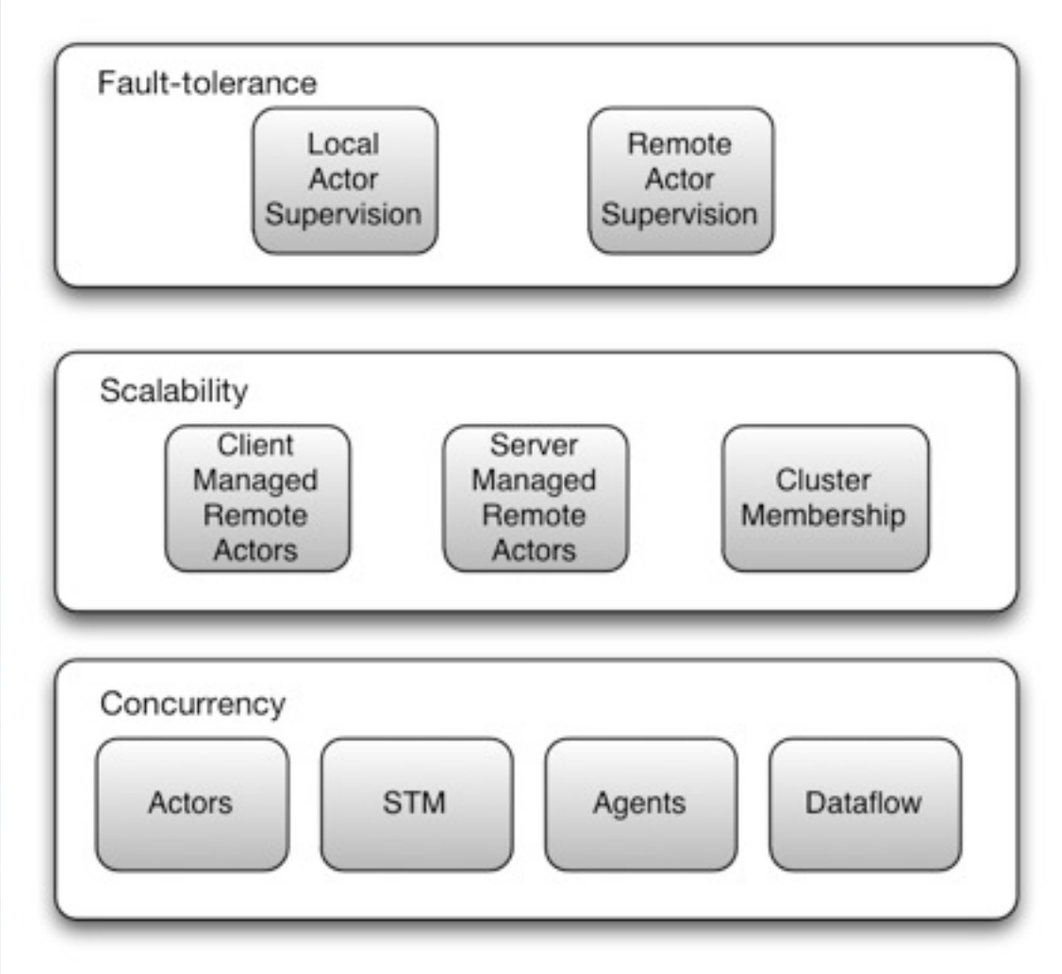
Embrace failure. Write applications that self-heal using Erlang-style Actor supervisor hierarchies.

Transparent Remoting

Remote Actors gives you a high-performance transparent distributed programming model.

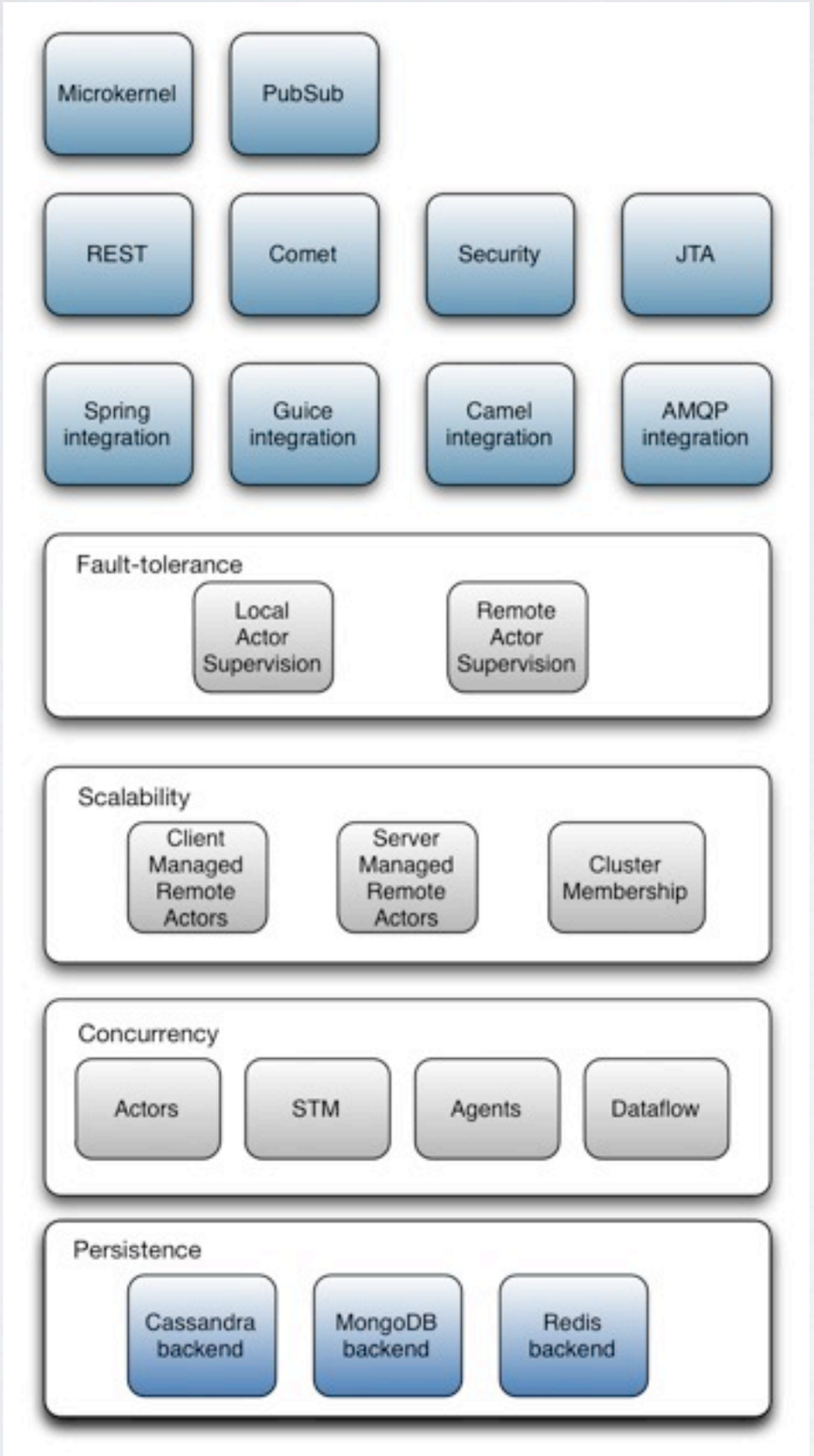
Java & Scala API

ARCHITECTURE



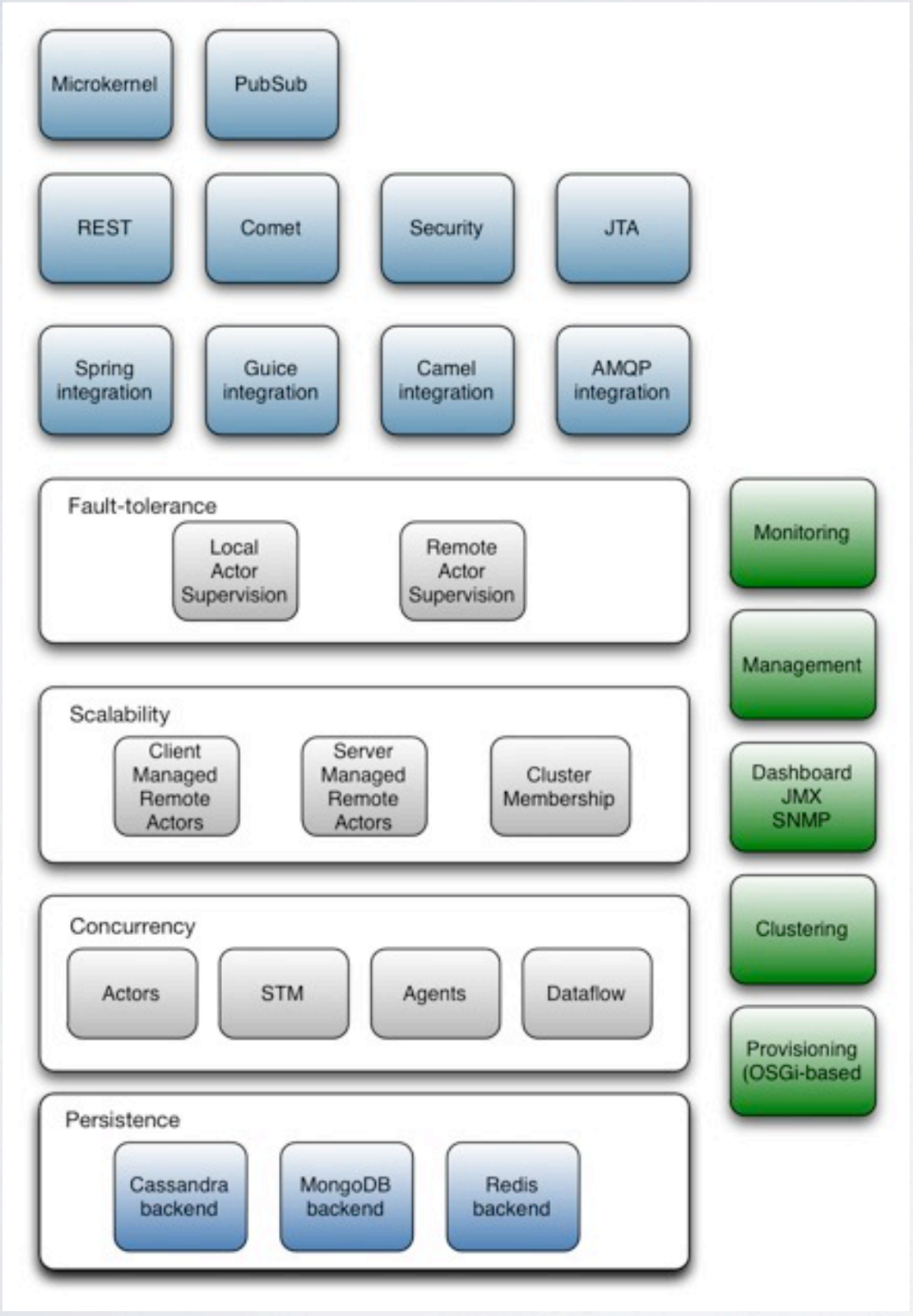
CORE SERVICES

ARCHITECTURE



ADD-ON MODULES

ARCHITECTURE



ENTERPRISE MODULES

Actors

one tool in the toolbox

Actor Model of Concurrency

- Implements Message-Passing Concurrency
- Share NOTHING
- Isolated lightweight processes
- Communicates through messages
- Asynchronous and non-blocking
- Each actor has a mailbox (message queue)

Actor Model of Concurrency

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

Akka Actors

Two different models

- Thread-based
- Event-based
  - Very lightweight (600 bytes per actor)
  - Can easily create millions on a single workstation (6.5 million on 4 G RAM)
  - Does not consume a thread

Actor

Actors

```
case object Tick

class Counter extends Actor {
  private var counter = 0

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

Create Actors

```
import Actor._

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
```

init & shutdown callbacks

```
class MyActor extends Actor {

  override def init = {
    ... // called after ‘start’
  }

  override def shutdown = {
    ... // called before ‘stop’
  }
}
```

the self reference

```
class RecursiveActor extends Actor {
  private var counter = 0
  self.id = “service:recursive”

  def receive = {
    case Tick =>
      counter += 1
      self ! Tick
  }
}
```

Actors

anonymous

```
val worker = actor {
  case Work(fn) => fn()
}
```

Send: !

```
counter ! Tick
```

fire-forget

Send: !!

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

uses Future under the hood (with time-out)

Send: !!

```
val resultOption = actor !! Message

val result = resultOption
               .getOrElse(defaultResult)

val result = resultOption.getOrElse(
    throw new Exception("Timed out")))
```

returns an Option[ResultType] directly

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

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
  }
}
```

UntypedActor

Untyped Actors

```
public class MyUntypedActor
  extends UntypedActor {

  public void onReceive(Object message)
    throws Exception {
    if (message instanceof String) {
      String msg = (String)message;
      System.out.println(
        "Received message: " + msg);
    }
  }
}
```

Create UntypedActor

```
UntypedActorRef actor =
  UntypedActor.actorOf(MyUntypedActor.class);

actor.start();
```

the context reference

```
class RecursiveActor
  extends UntypedActor {
  getContext().setId(“service:recursive”);

  public void onReceive(Object message)
    throws Exception {
    if (message instanceof Tick)
      getContext().sendOneWay(Tick);
  }
}
```

sendOneWay

```
actor.sendOneWay(msg);
```

fire-forget

sendRequestReply

```
Object res = actor.sendRequestReply(msg);
```

uses Future under the hood (with time-out)

sendRequestReplyFuture

```
// returns a future
Future future =
    actor.sendRequestReplyFuture(msg);

future.await();
Object result = future.get();
```

returns the Future directly

reply

```
public class HelloUntypedActor
  extends UntypedActor {

  public void onReceive(Object message)
    throws Exception {
    if (message instanceof String) {
      String name = (String)message;
      // use reply
      getContext().reply("Hi " + name);
    }
  }
}
```

TypedActor

Typed Actors

```
public class CounterImpl
   extends TypedActor implements Counter {

   private int counter = 0;

   public void count() {
      counter++;
      System.out.println(counter);
   }
}
```

Create Typed Actor

```
Counter counter =
  (Counter)TypedActor.newInstance(
    Counter.class, CounterImpl.class, 1000);
```

Send message

```
counter.count();
```

fire-forget

Request Reply

```
int hits = counter.getNrOfHits();
```

uses Future under the hood (with time-out)

the context reference

```
class PingImpl extends TypedActor
   implements Ping {
   public void hit(int count) {
      Pong pong =
         (Pong)getContext().getSender();
      pong.hit(count++);
   }
}
```

Immutable messages

```
// define the case class
case class Register(user: User)

// create and send a new case class message
actor ! Register(user)

// tuples
actor ! (username, password)

// lists
actor ! List(“bill”, “bob”, “alice”)
```

Actors: config

```
akka {
  version = "0.9.1"
  time-unit = "seconds"
  actor {
    timeout = 5
    throughput = 5
  }
}
```

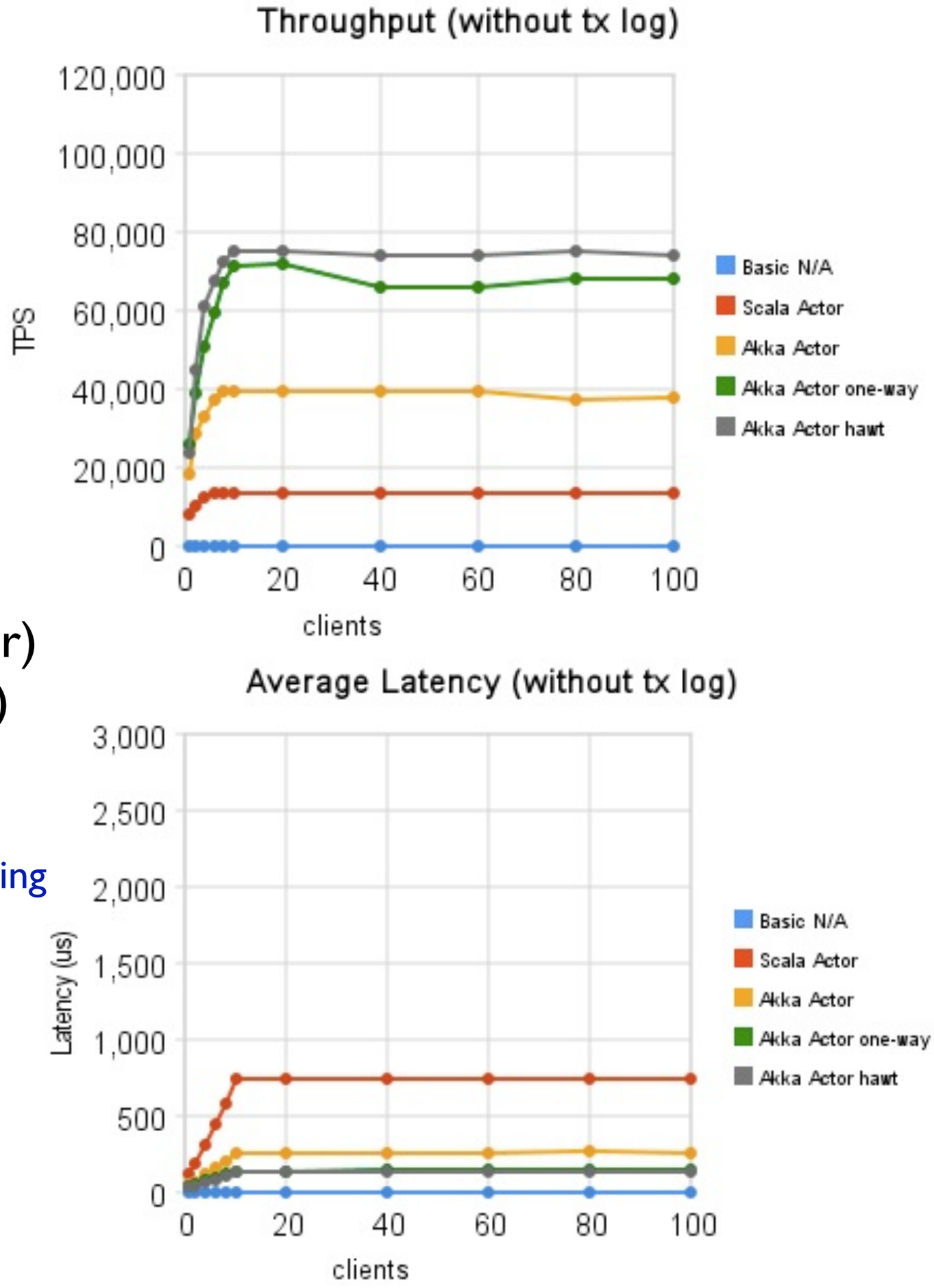
Scalability Benchmark

Simple Trading system

- Synchronous Scala version
- Scala Library Actors 2.8.0
- Akka request-reply
- Akka fire-forget (default dispatcher)
- Akka fire-forget (Hawt dispatcher)

Run it yourself:

http://github.com/patriknw/akka-sample-trading



Performance Microbenchmark

- Chameneos benchmark
  - http://shootout.alioth.debian.org/

Akka 0.8.1	**3815**
Scala Actors 2.8.0 (react)	**16086**

- +4 times faster
- Run it yourself (3 different benchmarks):
  - http://github.com/jboner/akka-bench

Agents

yet another tool in the toolbox

Agents

```
val agent = Agent(5)

// send function asynchronously
agent send (_ + 1)

val result = agent() // deref
... // use result

agent.close
```

Cooperates with STM

Akka Dispatchers

Dispatchers

Executor-based Dispatcher
Executor-based Work-stealing Dispatcher
Hawt Dispatcher
Reactor-based Dispatcher
Reactor-based Single-thread Dispatcher
Thread-based Dispatcher

Dispatchers

```
object Dispatchers {
  object globalHawtDispatcher extends HawtDispatcher
  ...

  def newExecutorBasedEventDrivenDispatcher(name: String)

  def newExecutorBasedEventDrivenWorkStealingDispatcher(name: String)

  def newHawtDispatcher(aggregate: Boolean)

  ...
}
```

Set dispatcher

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

  ...
}

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

EventBasedDispatcher

Fluent DSL

```
val dispatcher =
  Dispatchers.newExecutorBasedEventDrivenDispatcher
dispatcher
    .withNewThreadPoolWithBoundedBlockingQueue(100)
    .setCorePoolSize(16)
    .setMaxPoolSize(128)
    .setKeepAliveTimeInMillis(60000)
    .setRejectionPolicy(new CallerRunsPolicy)
    .buildThreadPool
```

MessageQueues

Unbounded LinkedBlockingQueue

Bounded LinkedBlockingQueue

Bounded ArrayBlockingQueue

Bounded SynchronousQueue

Plus different options per queue

Let it crash

fault-tolerance

Stolen from

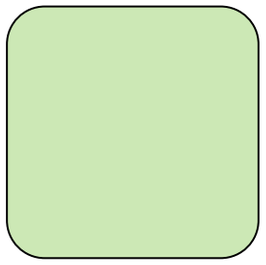
Erlang

9

nines

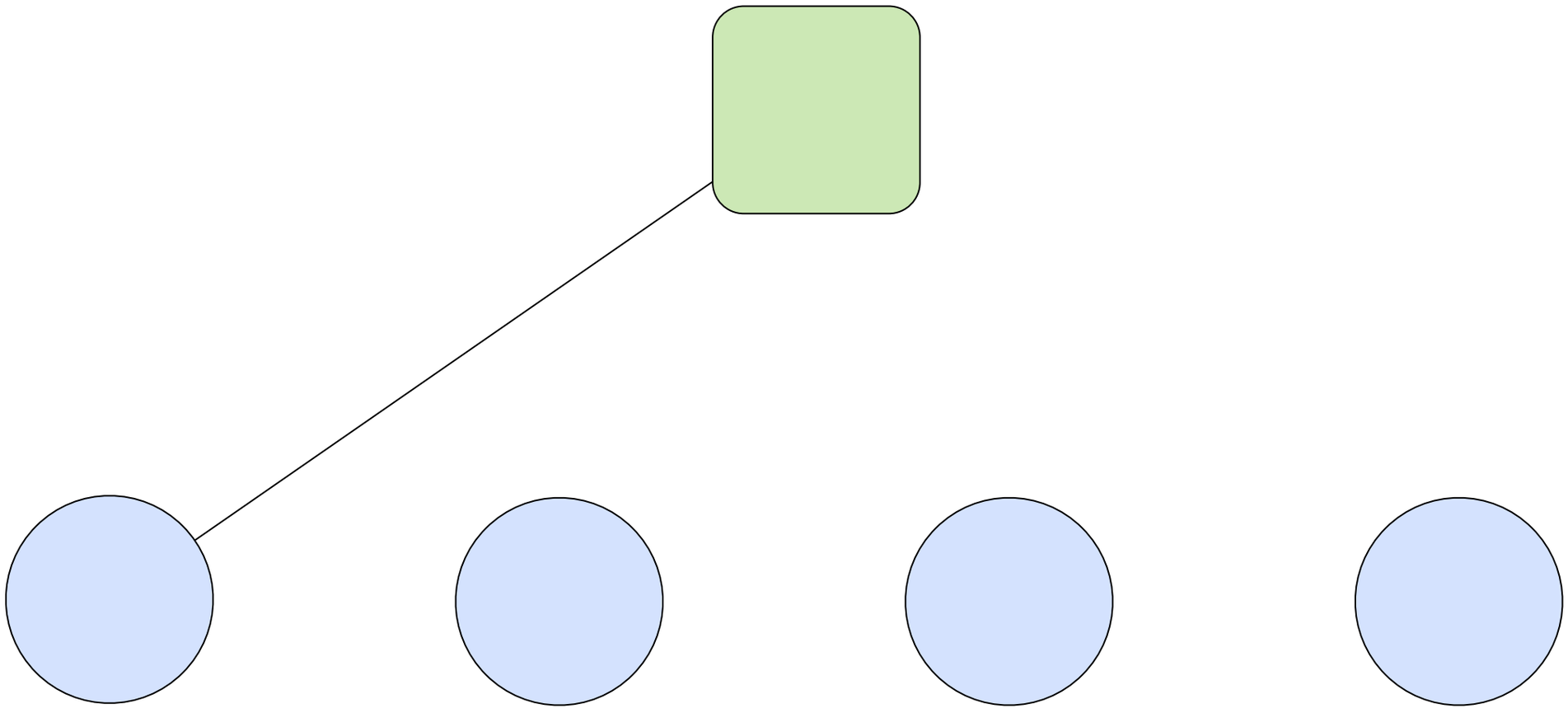
OneForOne

fault handling strategy



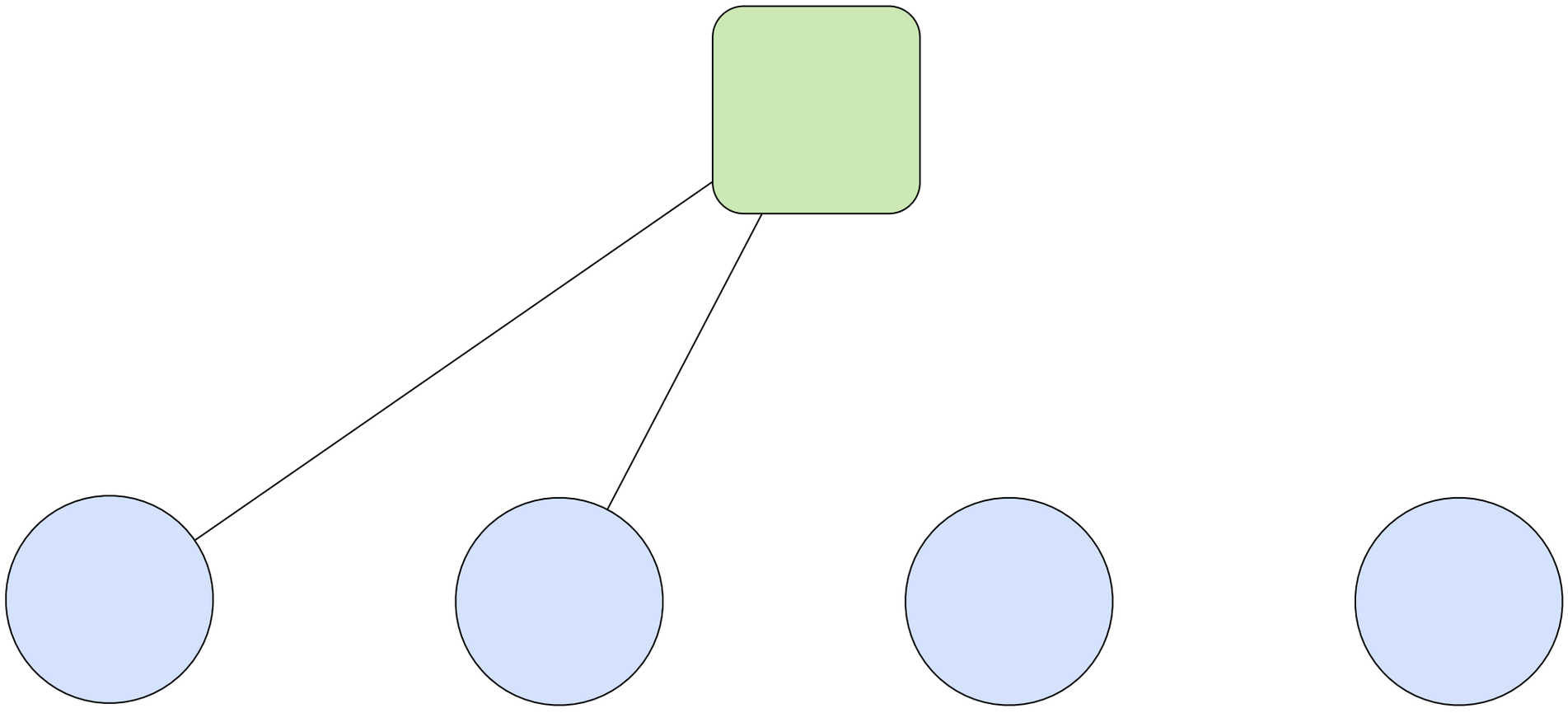
OneForOne

fault handling strategy



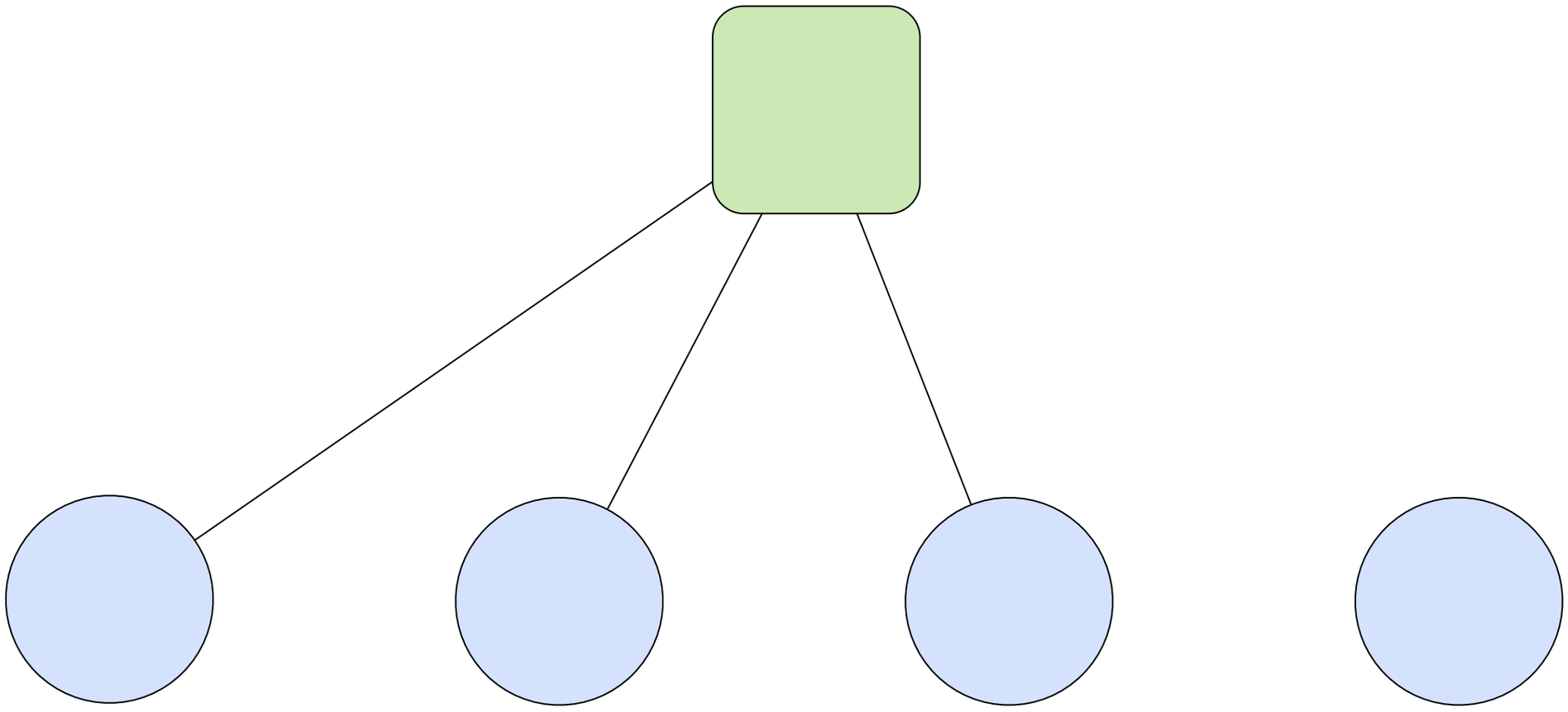
OneForOne

fault handling strategy



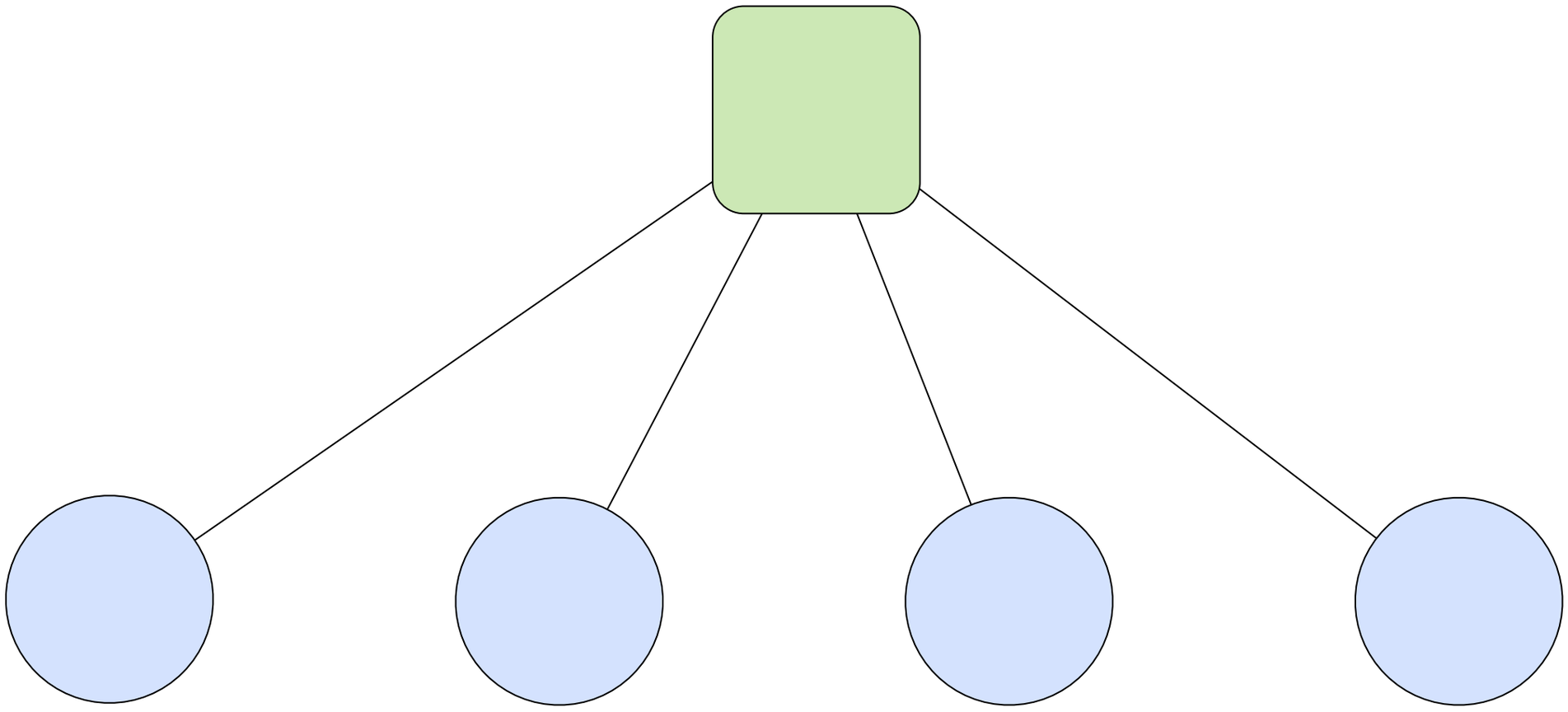
OneForOne

fault handling strategy



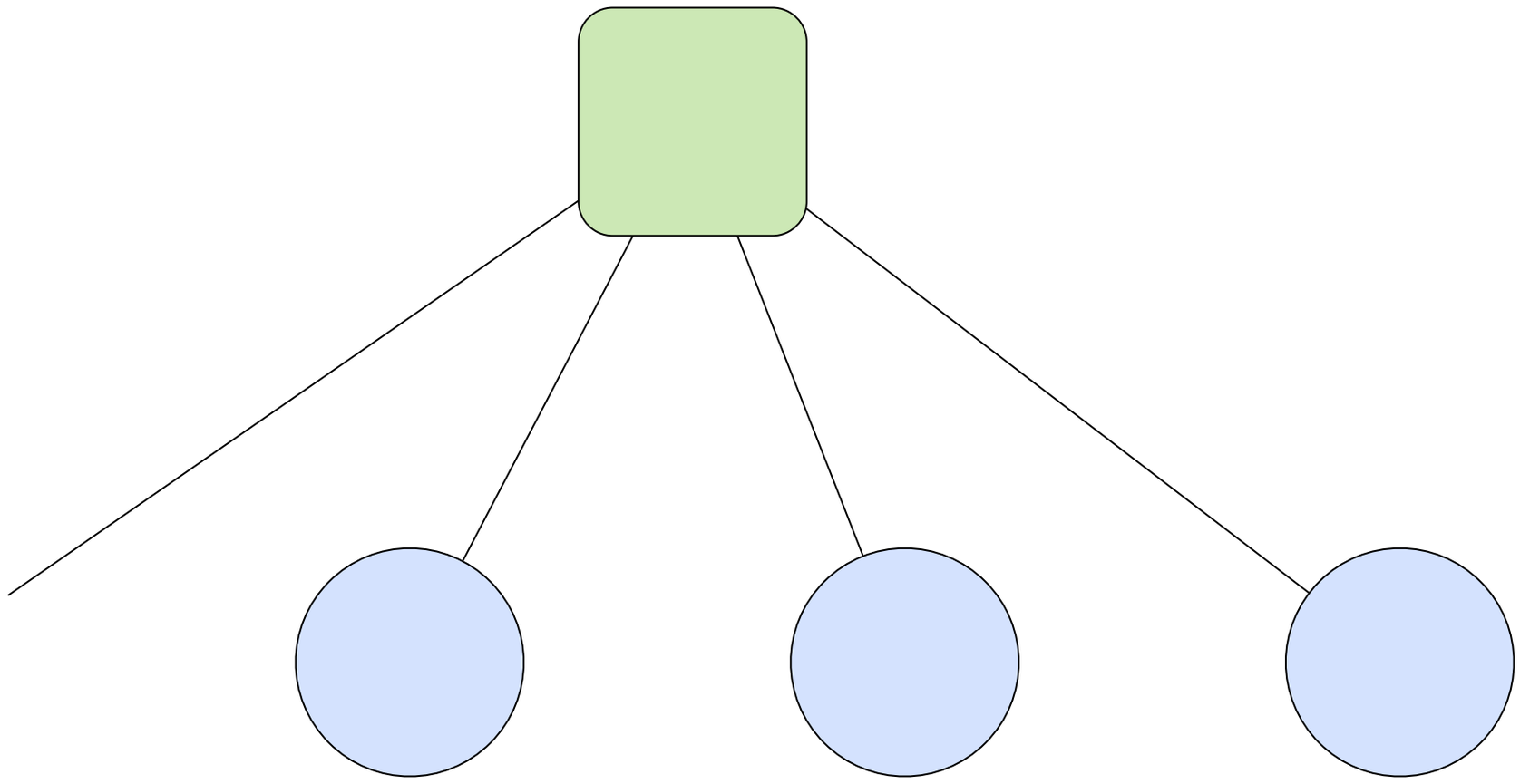
OneForOne

fault handling strategy



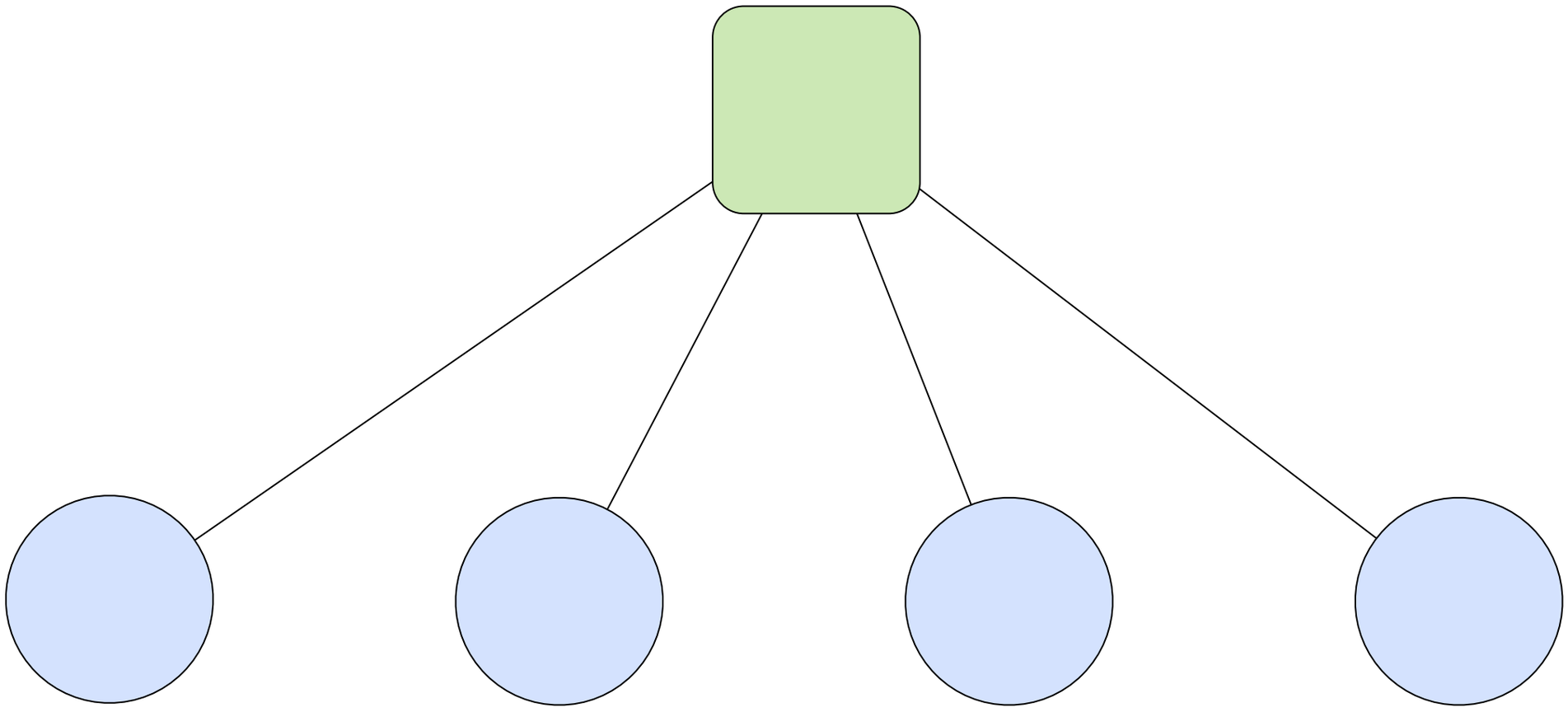
OneForOne

fault handling strategy



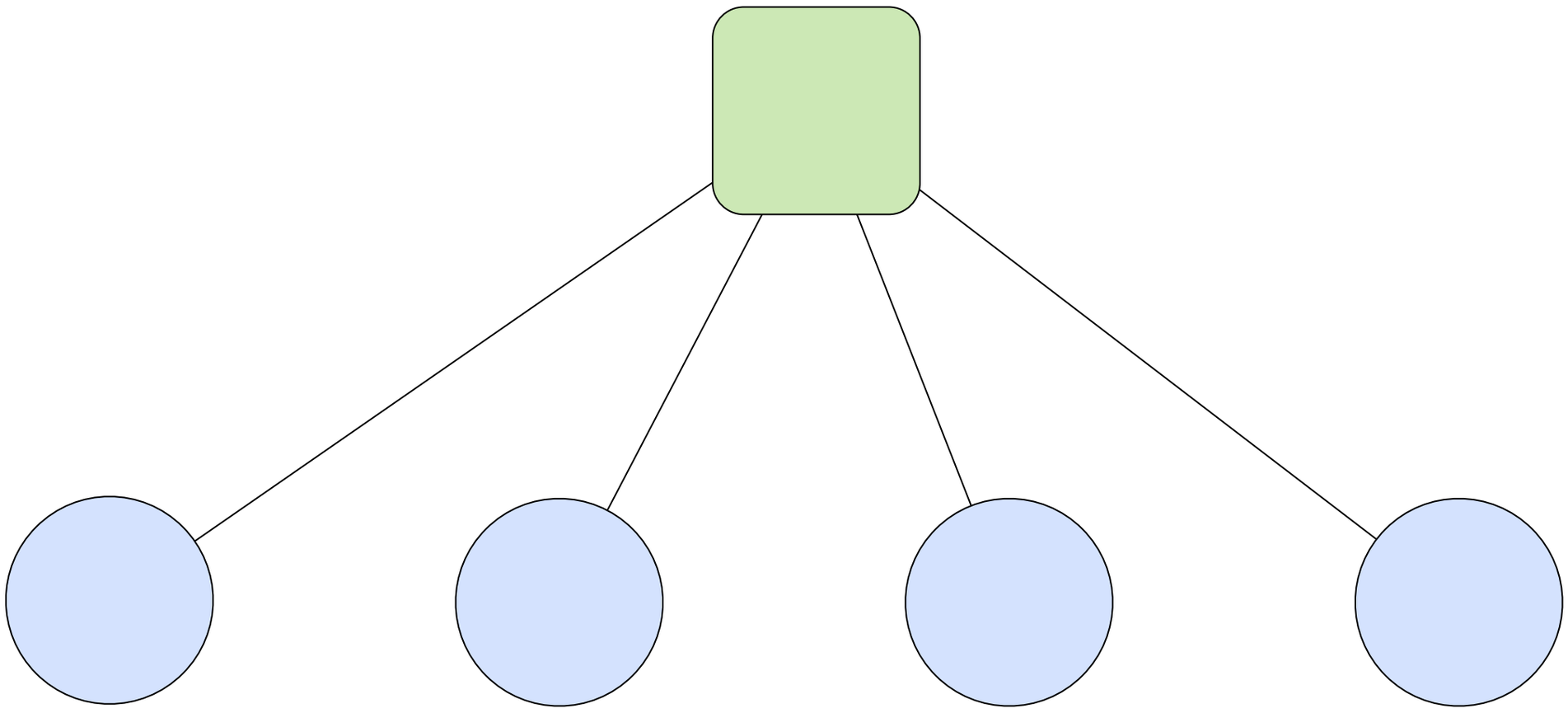
OneForOne

fault handling strategy



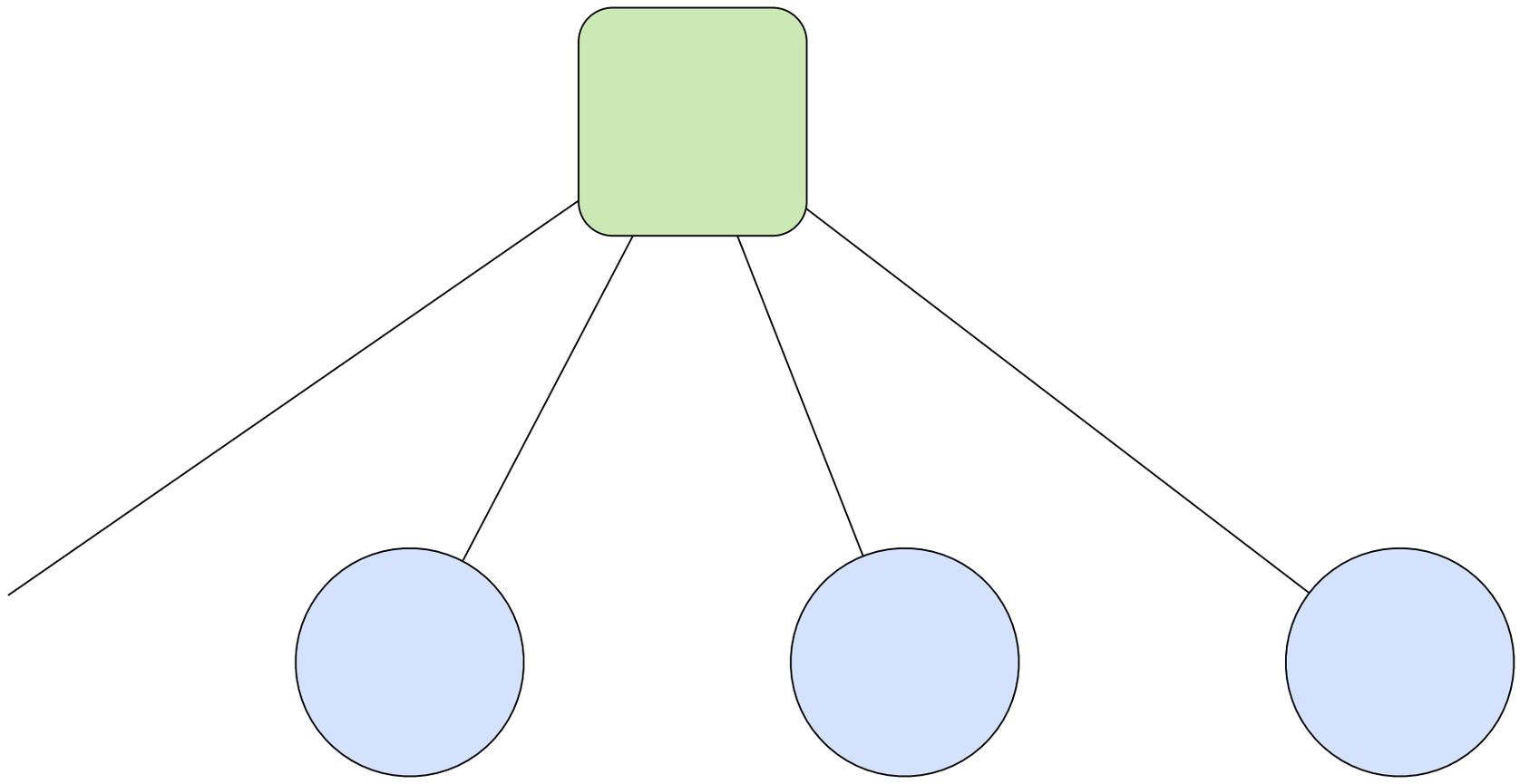
AllForOne

fault handling strategy



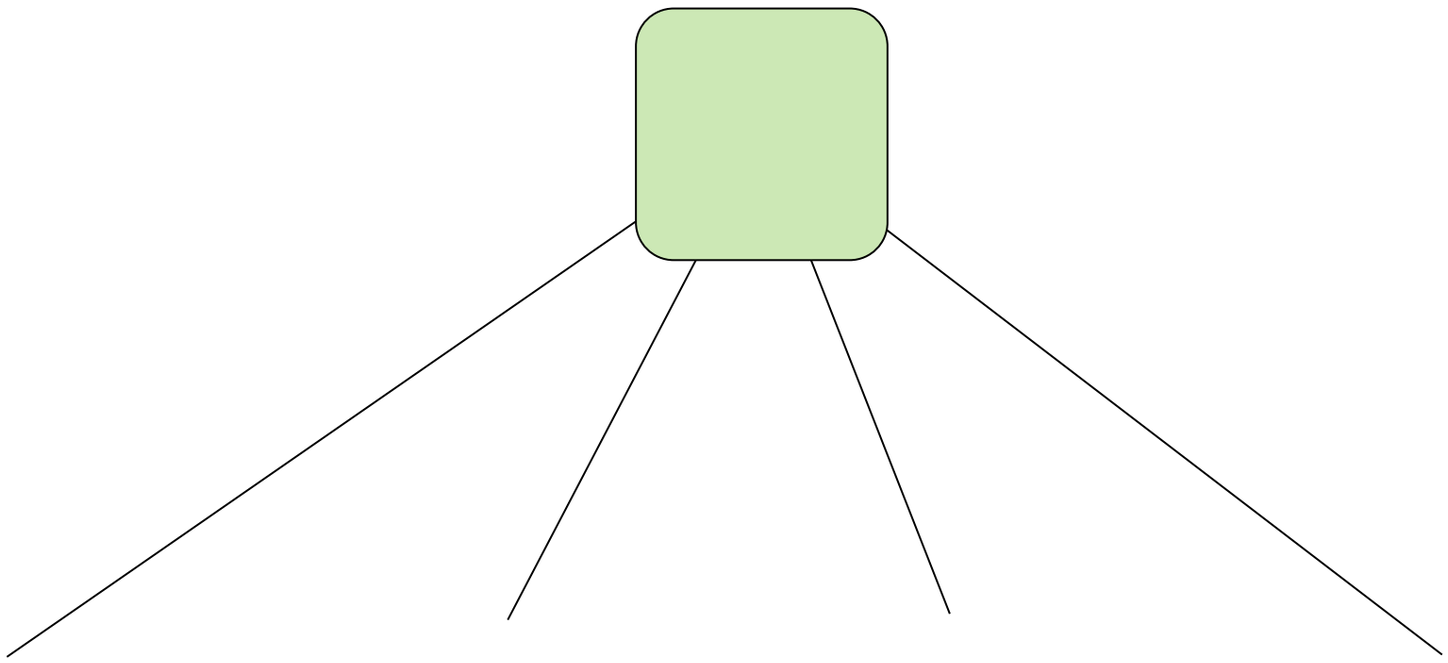
AllForOne

fault handling strategy



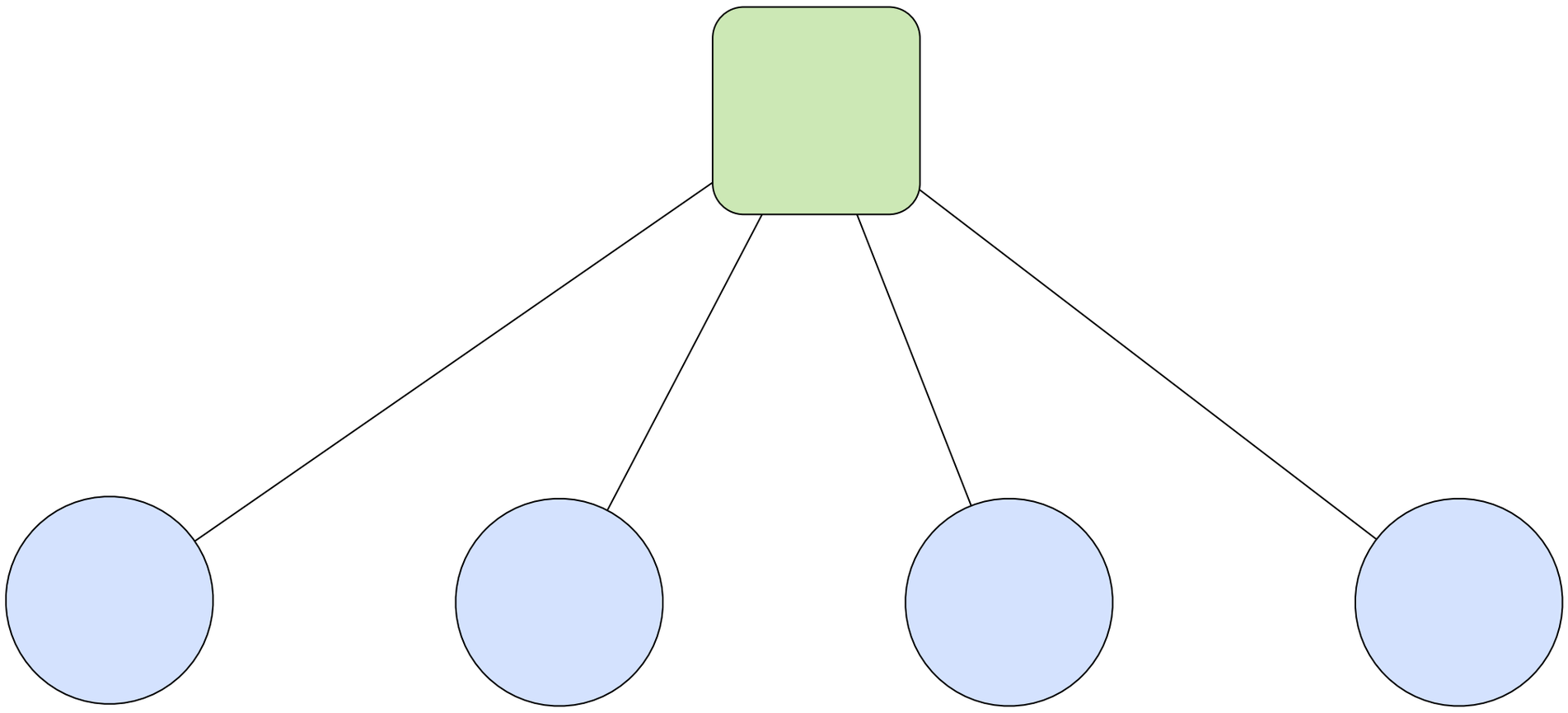
AllForOne

fault handling strategy

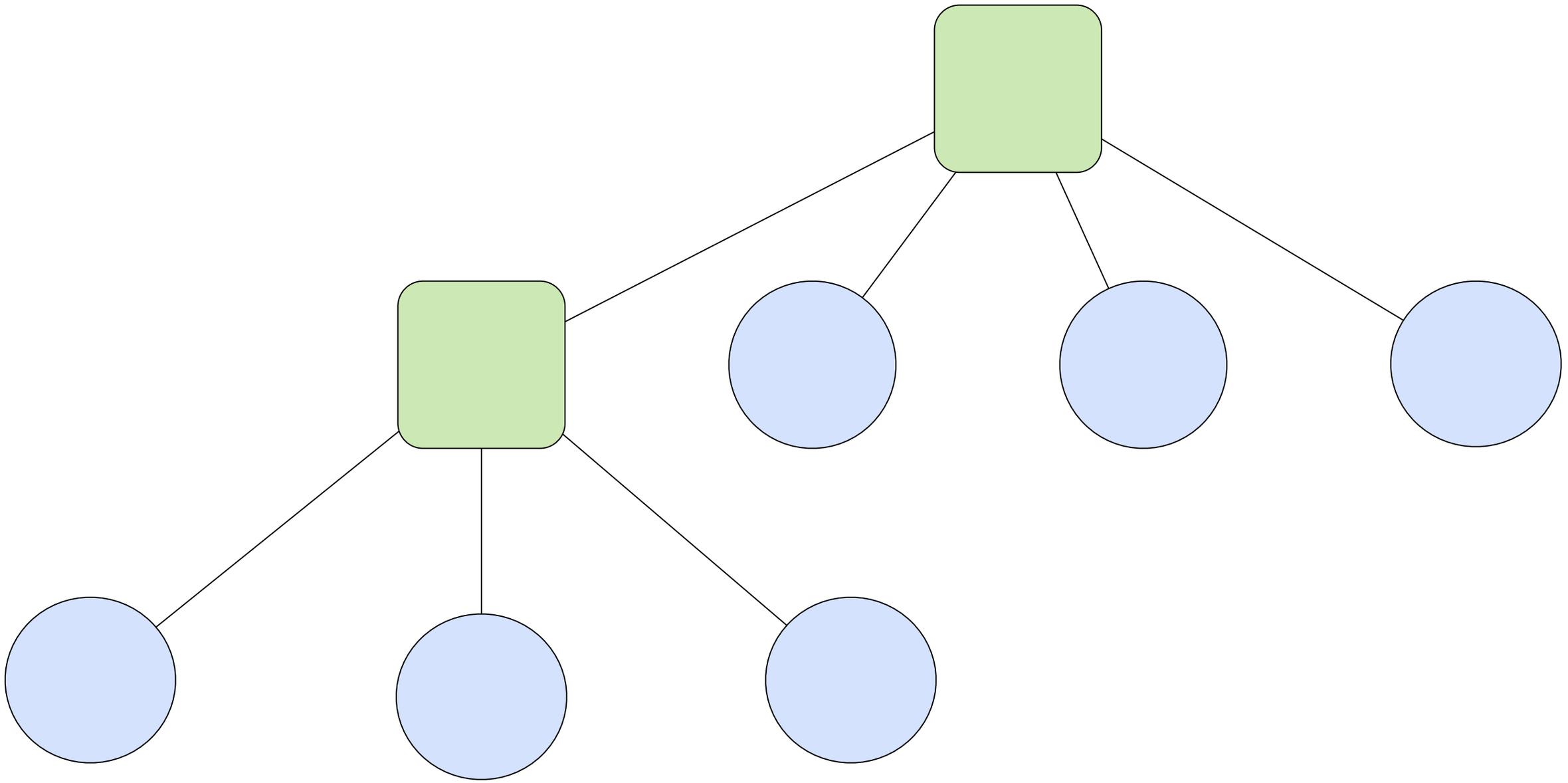


AllForOne

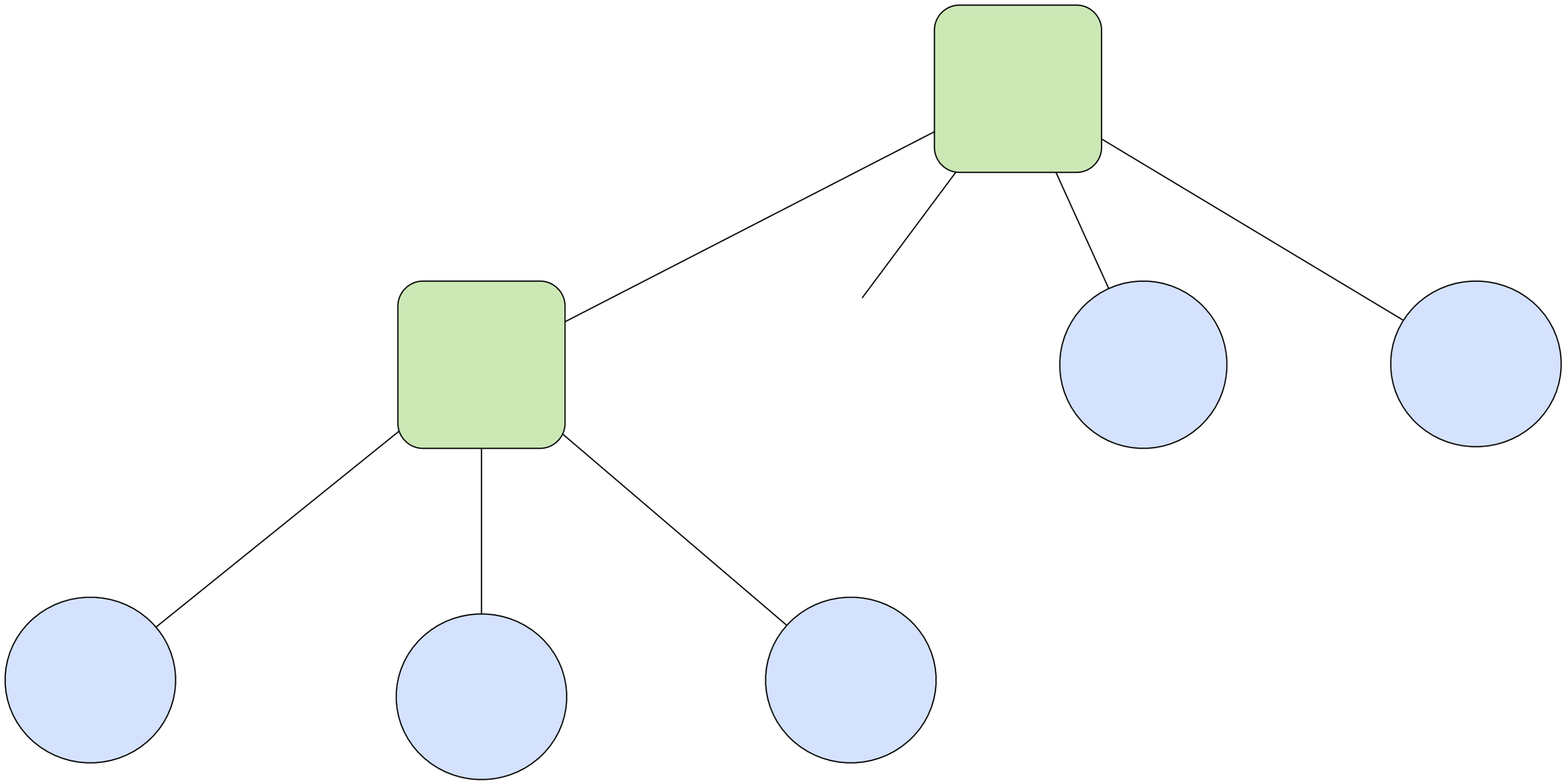
fault handling strategy



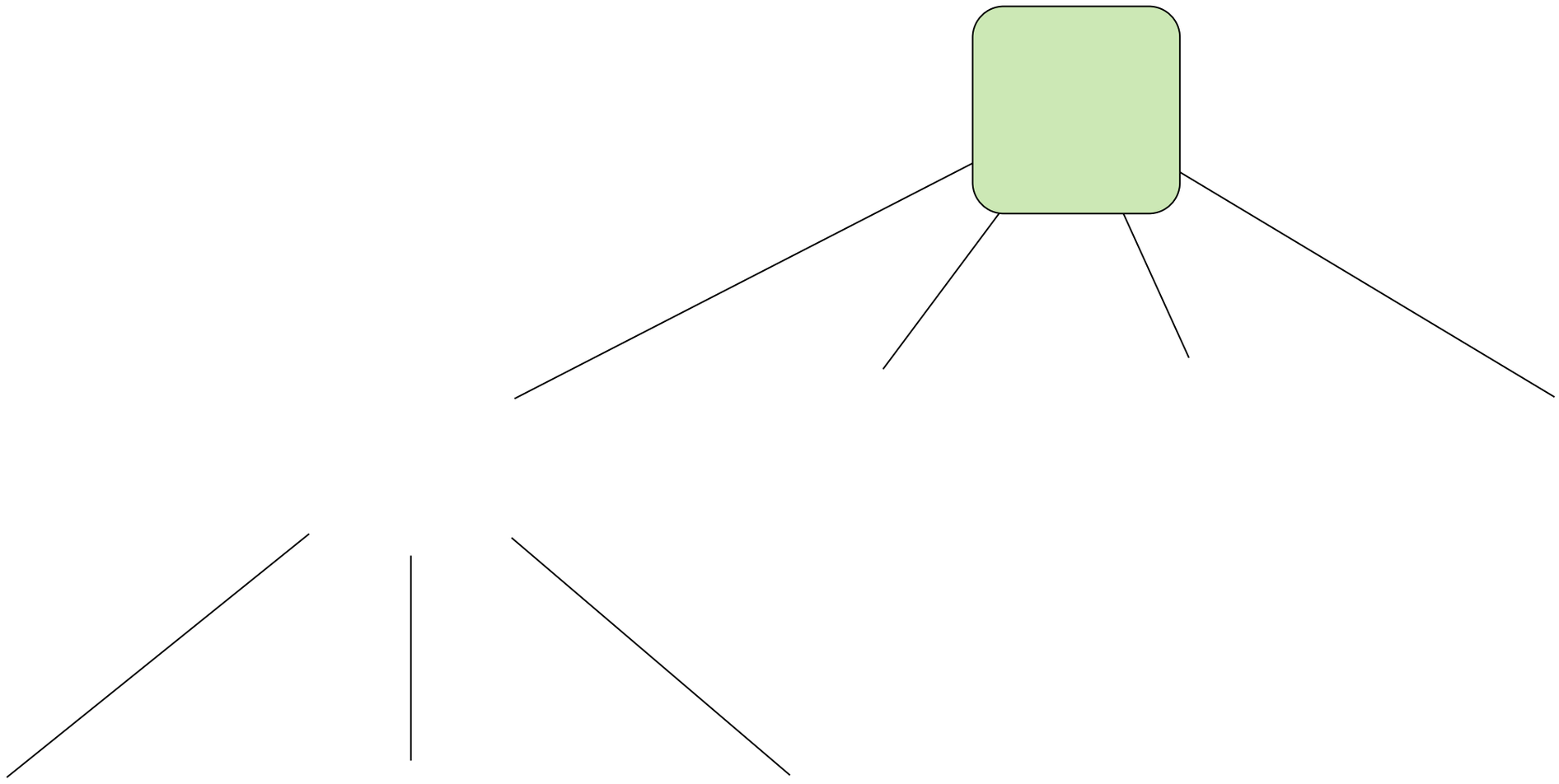
Supervisor hierarchies



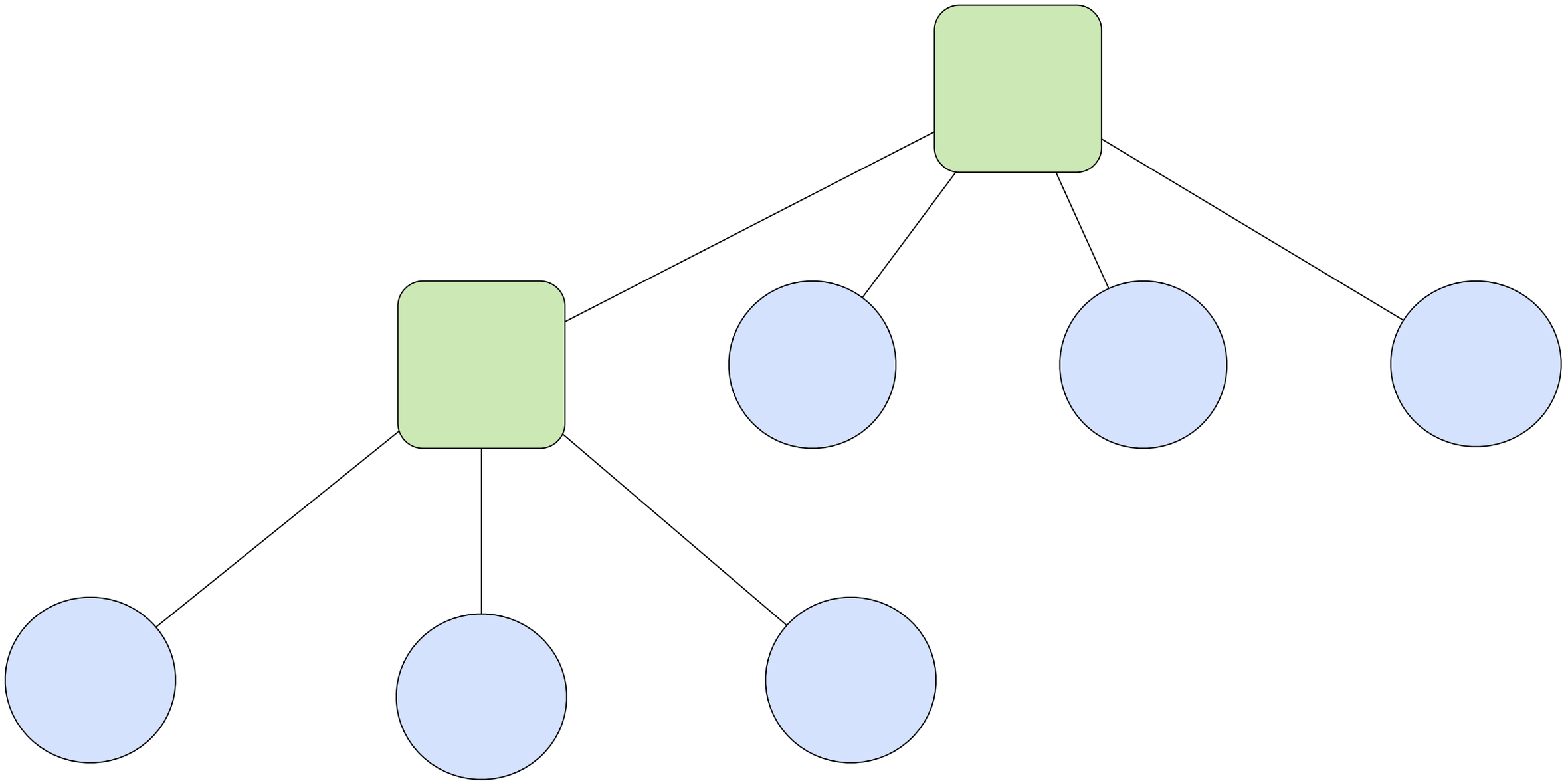
Supervisor hierarchies



Supervisor hierarchies



Supervisor hierarchies



Fault handlers

```
AllForOneStrategy(
  maxNrOfRetries,
  withinTimeRange)

OneForOneStrategy(
  maxNrOfRetries,
  withinTimeRange)
```

Linking

```
link(actor)
unlink(actor)

startLink(actor)
spawnLink[MyActor]
```

trapExit

```
trapExit = List(
  classOf[ServiceException],
  classOf[PersistenceException])
```

Supervision

```
class Supervisor extends Actor {
  trapExit = List(classOf[Throwable])
  faultHandler =
    Some(OneForOneStrategy(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

```
val supervisor = Supervisor(
  SupervisorConfig(
    RestartStrategy(AllForOne, 3, 10000),
     Supervise(
      actor1,
      LifeCycle(Permanent)) ::
     Supervise(
      actor2,
      LifeCycle(Temporary)) ::
    Nil))
```

ActorRegistry

```
val actors = ActorRegistry.actors
val actors = ActorRegistry.actorsFor[TYPE]
val actors = ActorRegistry.actorsFor(id)
val actor = ActorRegistry.actorFor(uuid)
ActorRegistry.foreach(fn)
ActorRegistry.shutdownAll
```

Remote Actors

Remote Server

```
// use host & port in config
RemoteNode.start

RemoteNode.start("localhost", 9999)
```

Scalable implementation based on
NIO (Netty) & Protobuf

Two types of

remote actors

- Client-initated and managed
- Server-initiated and managed

Client-managed

supervision works across nodes

```
// methods in Actor class

spawnRemote[MyActor](host, port)

spawnLinkRemote[MyActor](host, port)

startLinkRemote(actor, host, port)
```

Client-managed

moves actor to server
client manages through proxy

```
val actorProxy = spawnLinkRemote[MyActor](
  “darkstar”,
  9999)

actorProxy ! message
```

Server-managed

register and manage actor on server

client gets “dumb” proxy handle

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

server part

Server-managed

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

handle ! message
```

client part

Cluster Membership

```
Cluster.relayMessage(
  classOf[TypeOfActor], message)

for (endpoint <- Cluster)
  spawnRemote[TypeOfActor](
                endpoint.host,
                endpoint.port)
```

Remote config

```
akka {
  remote {
    service = on
    hostname = "homer.lan"
    port = 9999
    connection-timeout = 1000
  }
}
```

STM

yet another tool in the toolbox

What is STM?

Software Transactional Memory (STM)

STM: overview

- See the memory (heap and stack) as a transactional dataset
- Similar to a database
  - begin
  - commit
  - abort/rollback
- Transactions are retried automatically upon collision
- Rolls back the memory on abort

STM: overview

- Transactions can nest
- Transactions compose (yipee!!)

```
atomic {
    ..
    atomic {
        ..
    }
}
```

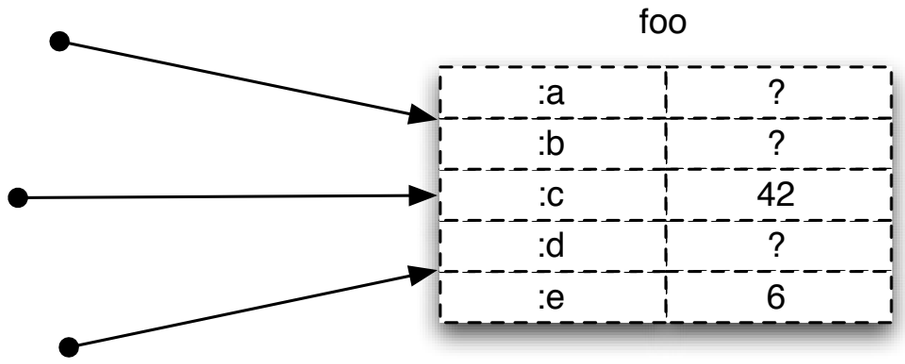
STM: restrictions

>All operations in scope of a transaction:

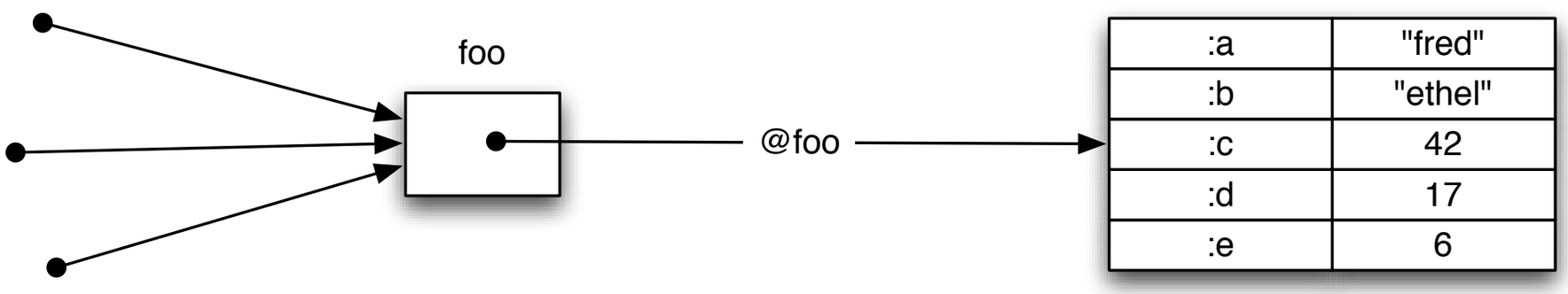
- Need to be idempotent
- Can't have side-effects

Managed References

- Typical OO: direct access to mutable objects



- Managed Reference: separates Identity & Value



Managed References

- Separates Identity from Value
  - Values are immutable
  - Identity (Ref) holds Values
- Change is a function
- Compare-and-swap (CAS)
- Abstraction of time
- Must be used within a transaction

Managed References

```
val ref = Ref(Map[String, User]())

val users = ref.get

val newUsers = users + (“bill” -> User(“bill”))

ref.swap(newUsers)
```

Transactional datastructures

```
val users = TransactionalMap[String, User]()

val users = TransactionalVector[User]()
```

atomic

```
import Transaction.Local._

atomic {
  ...
  atomic { // nested transactions compose
    ... // do something within a transaction
  }
}
```

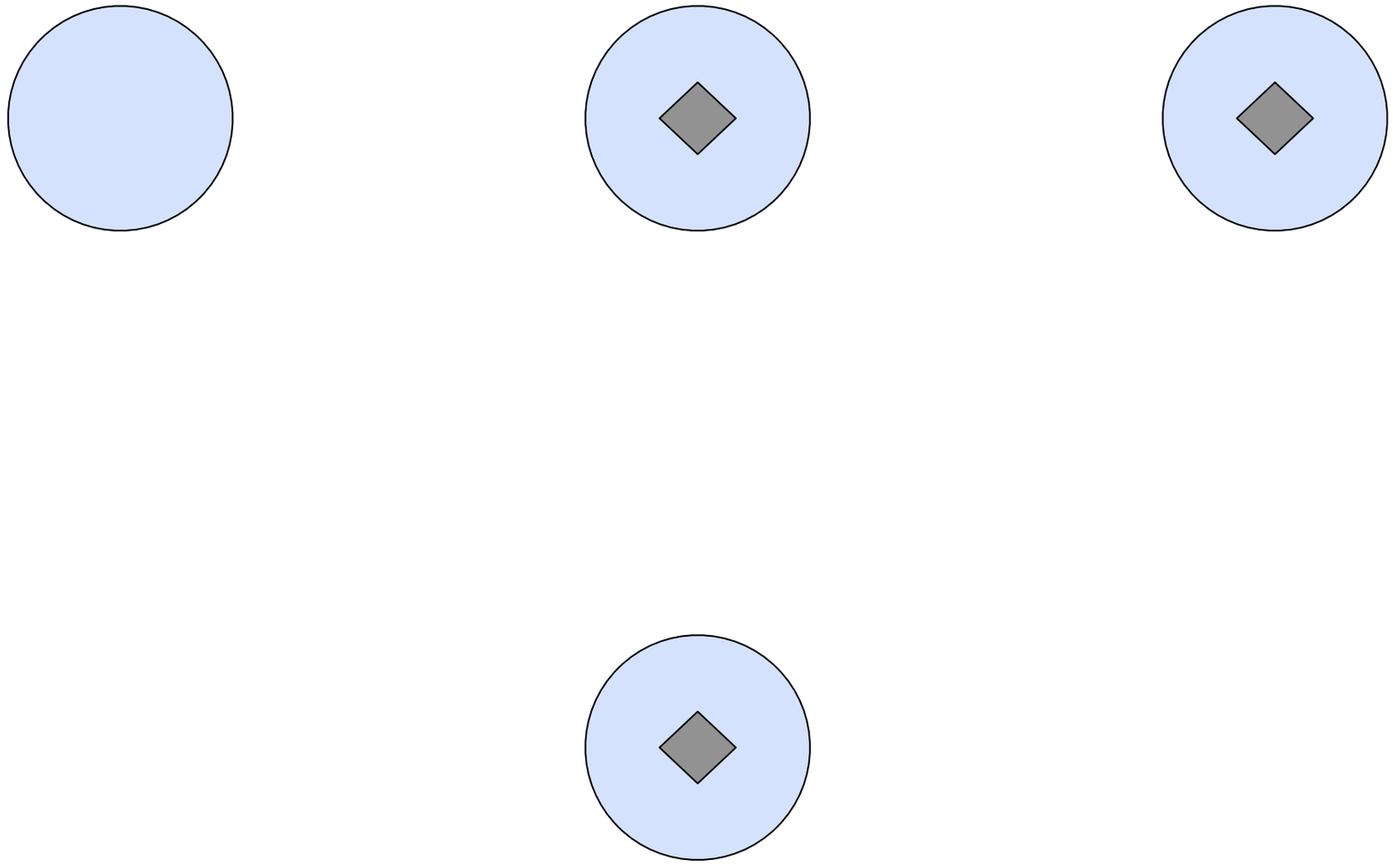
Actors + STM =

Transactors

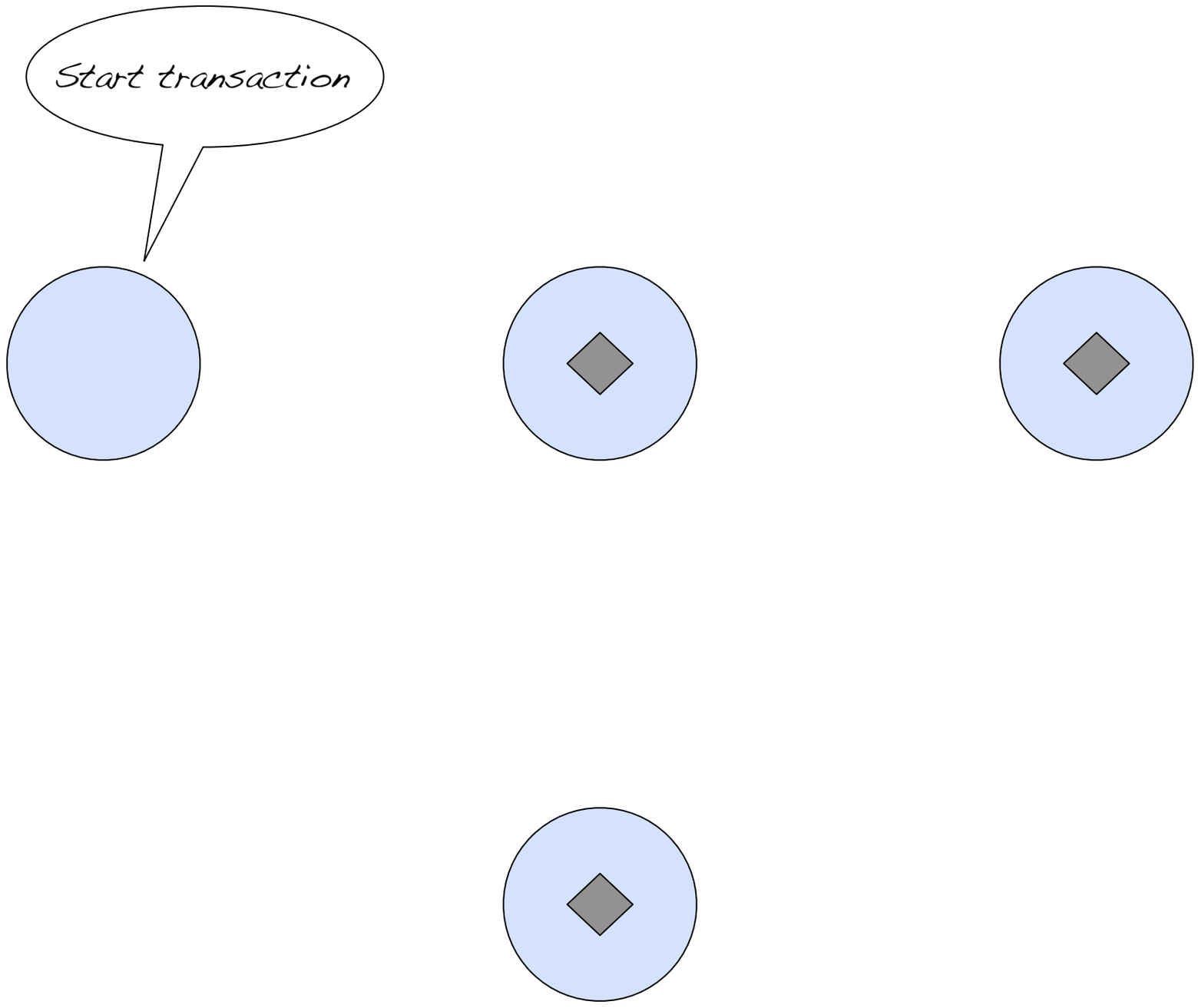
Transactors

```
class UserRegistry extends Transactor {

  private lazy val storage =
    TransactionalMap[String, User]()

  def receive = {
    case NewUser(user) =>
      storage + (user.name -> user)
    ...
  }
}
```

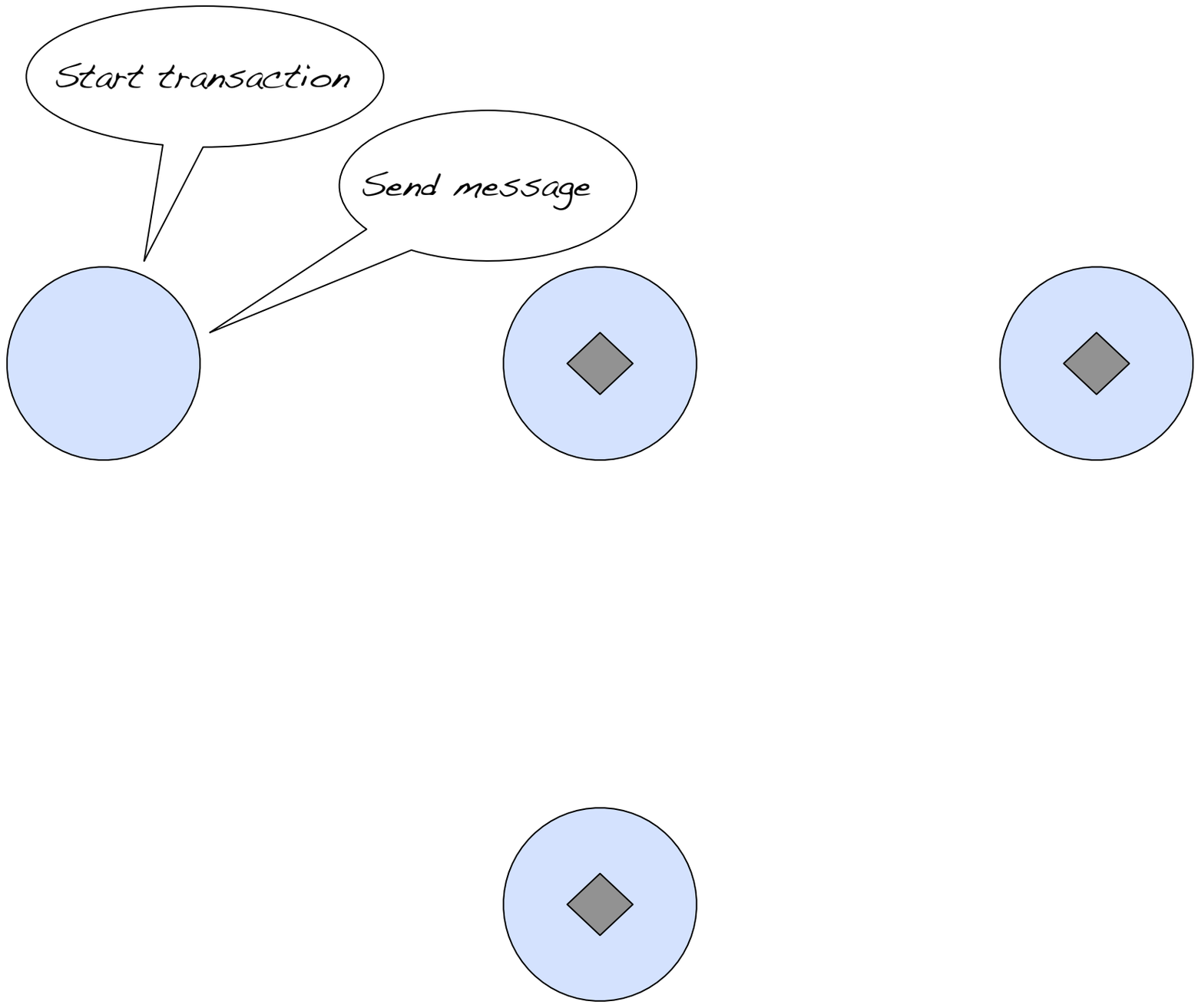
Transactors



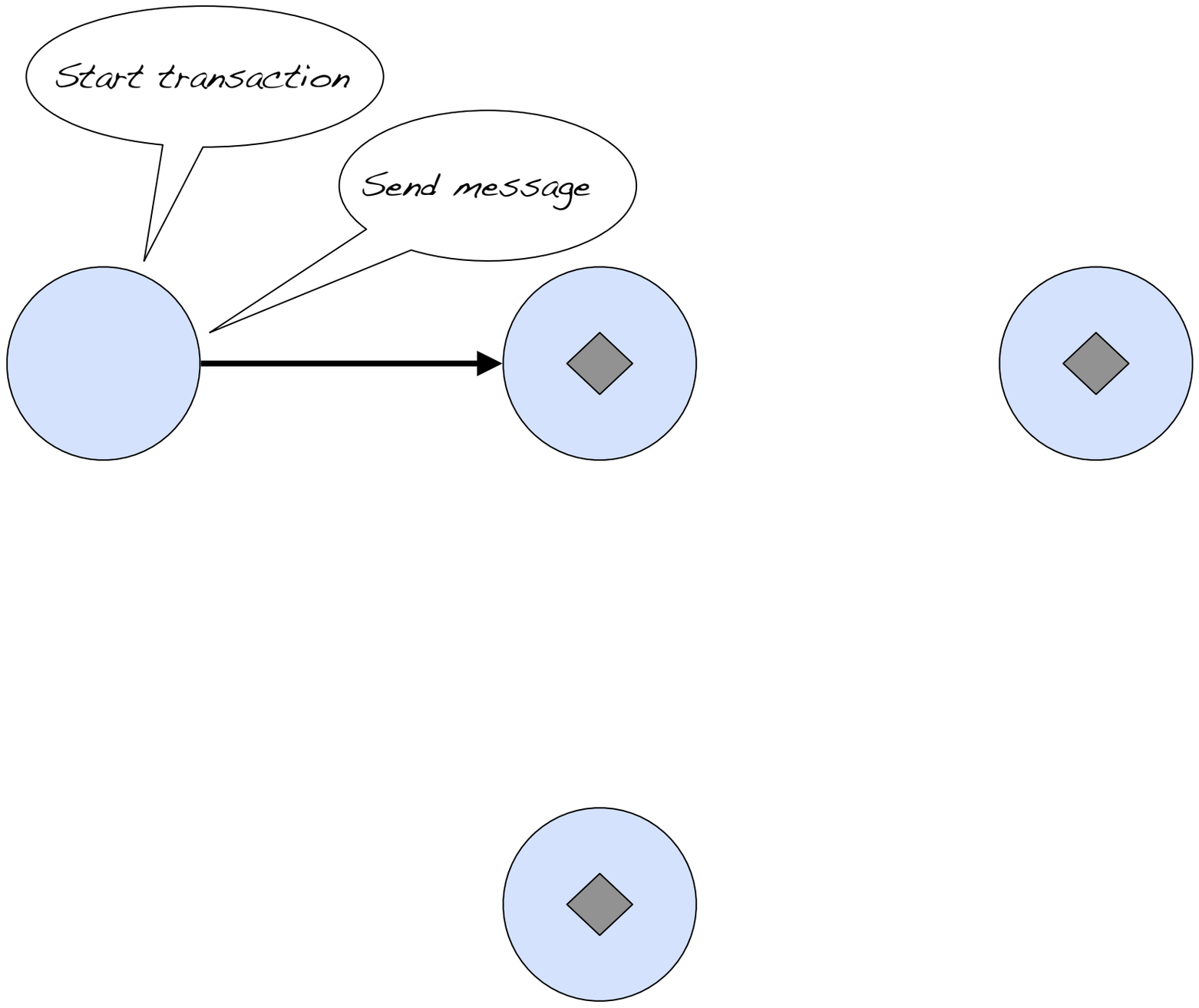
Transactors



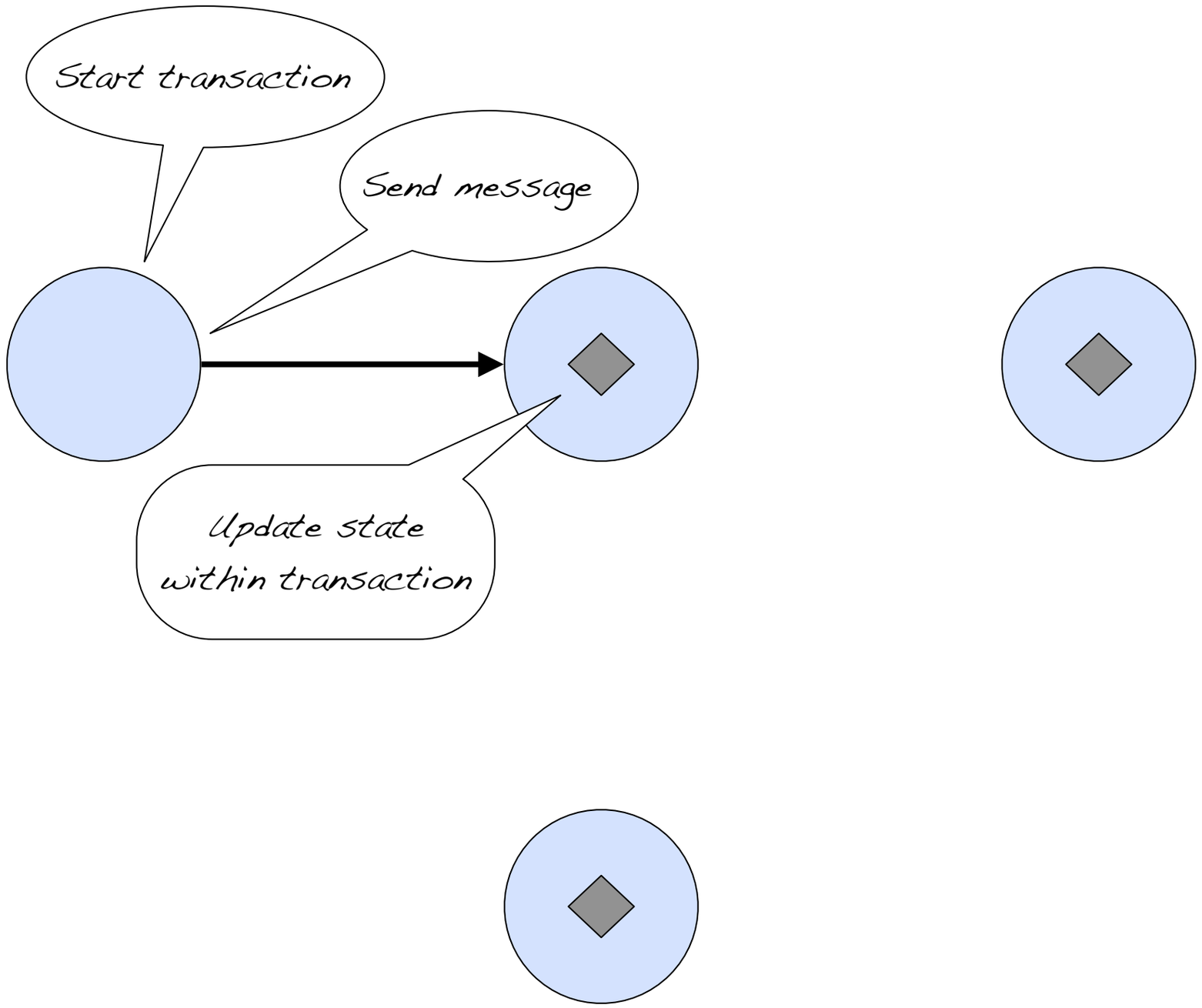
Transactors



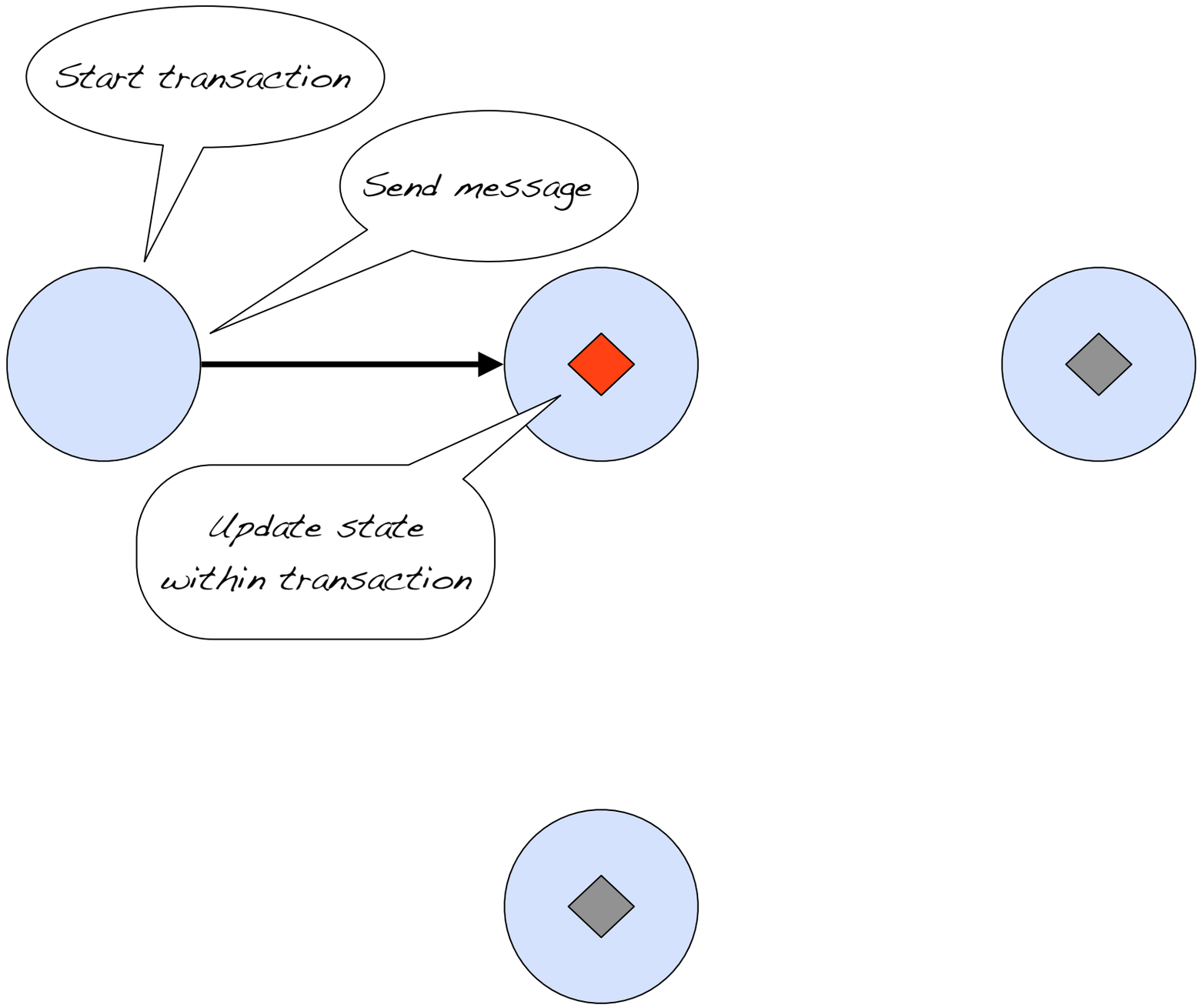
Transactors



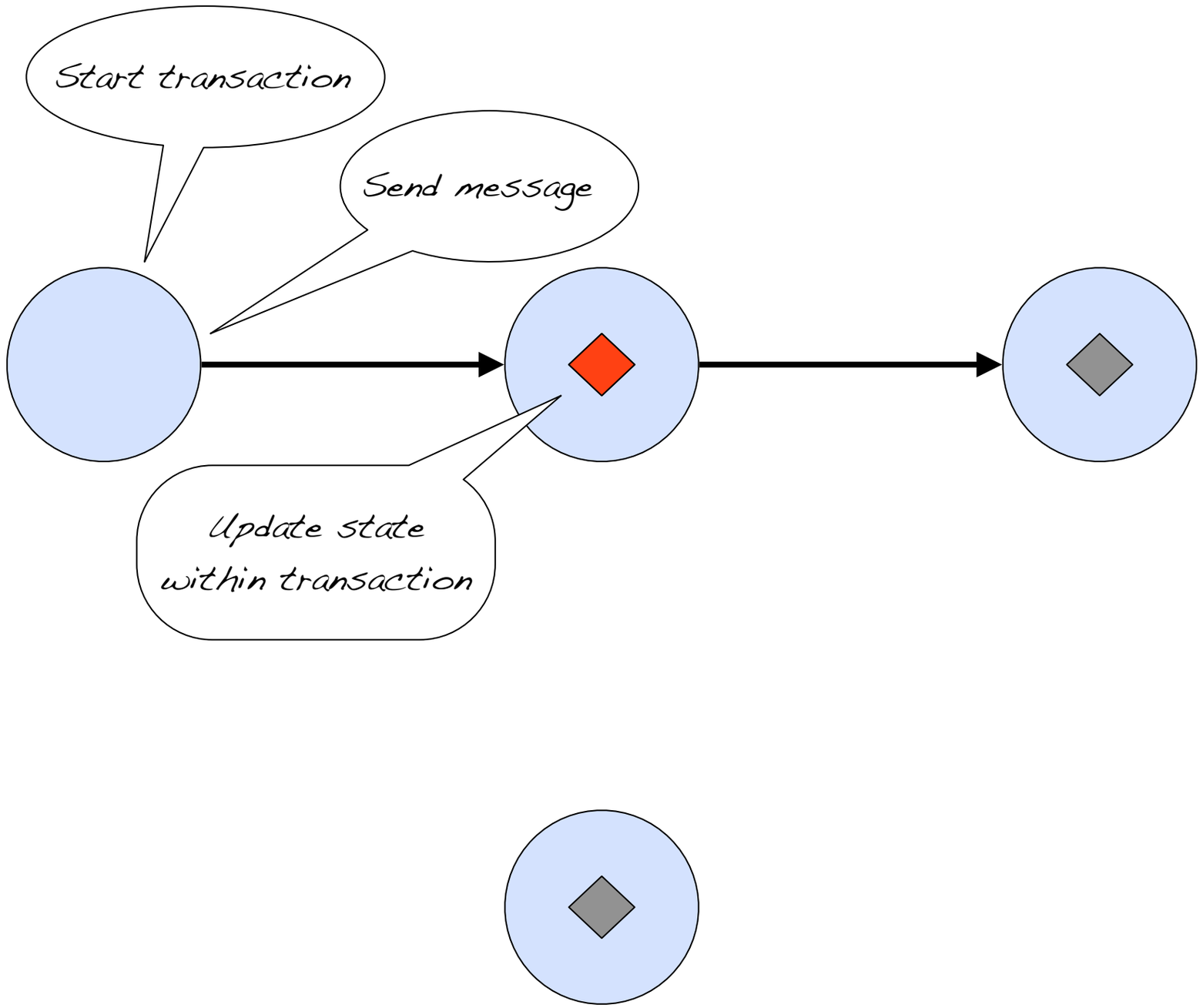
Transactors



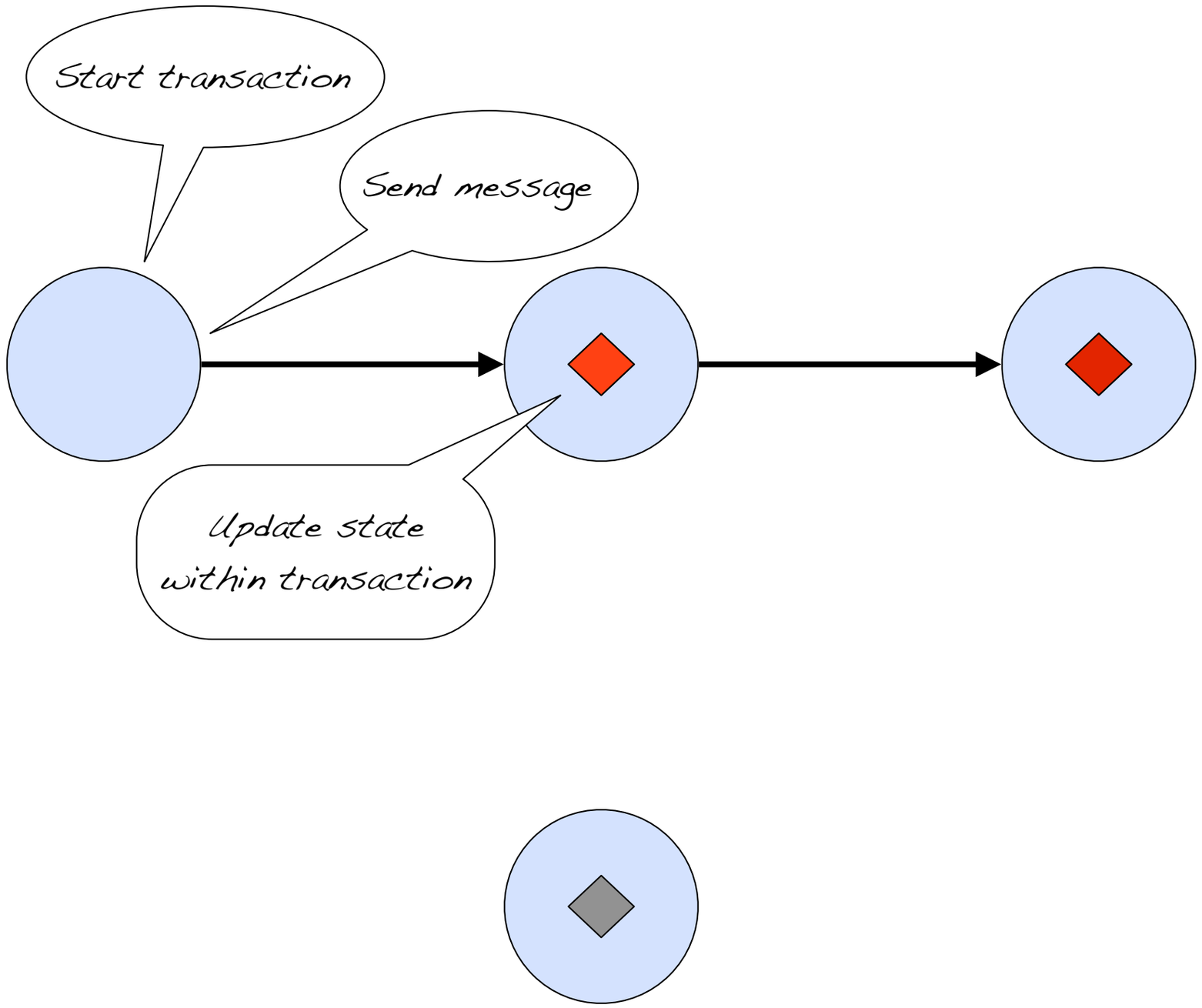
Transactors



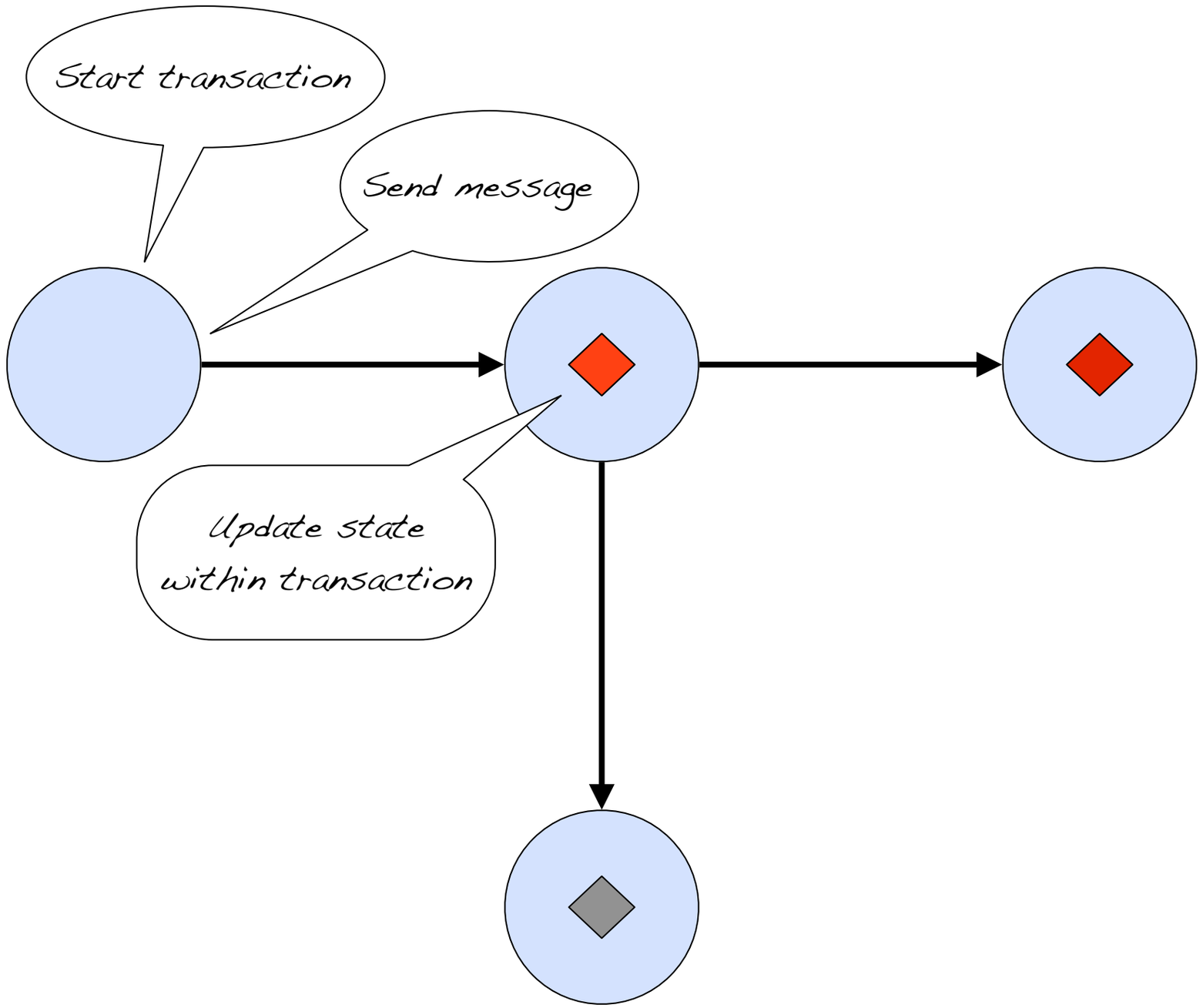
Transactors



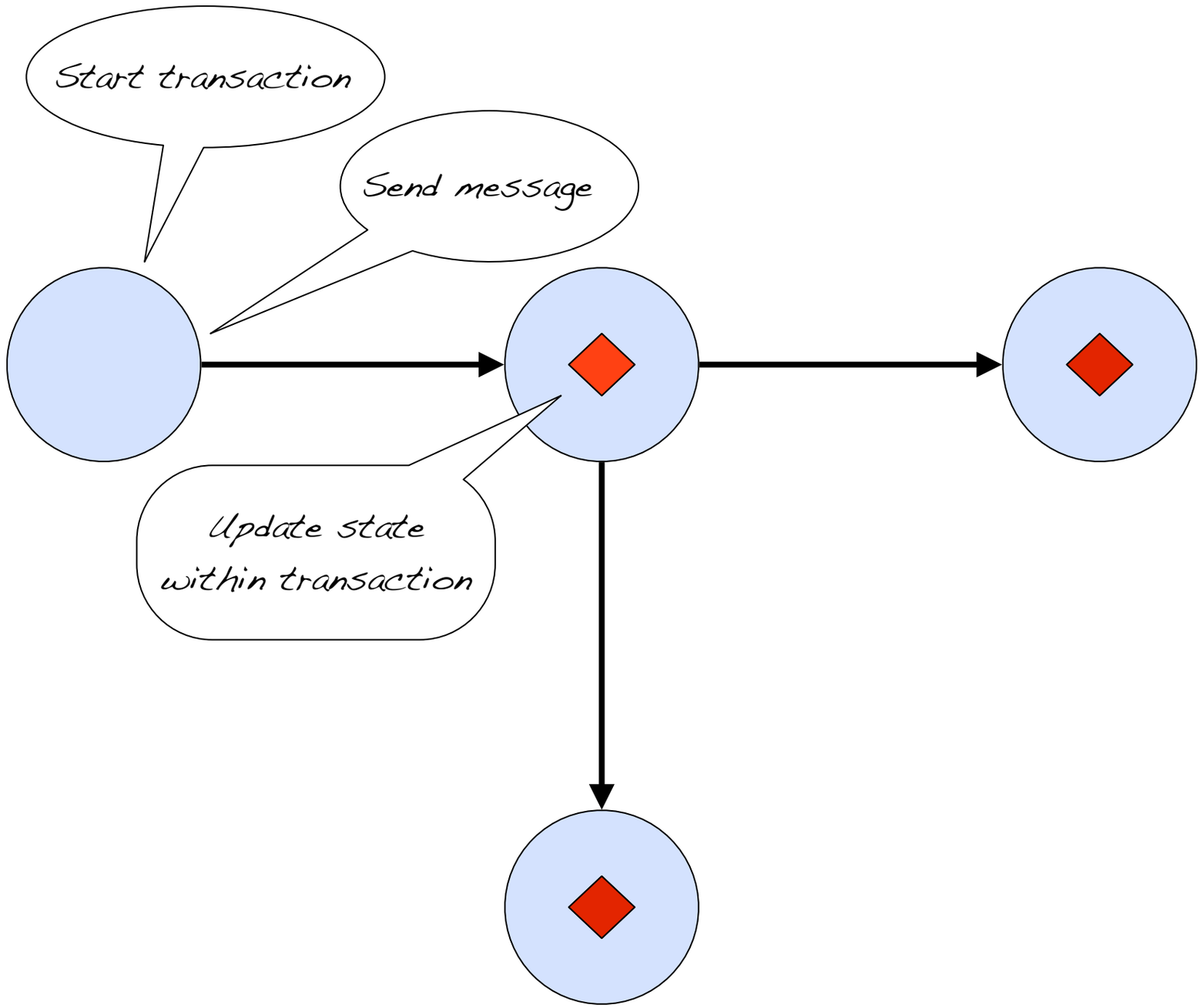
Transactors



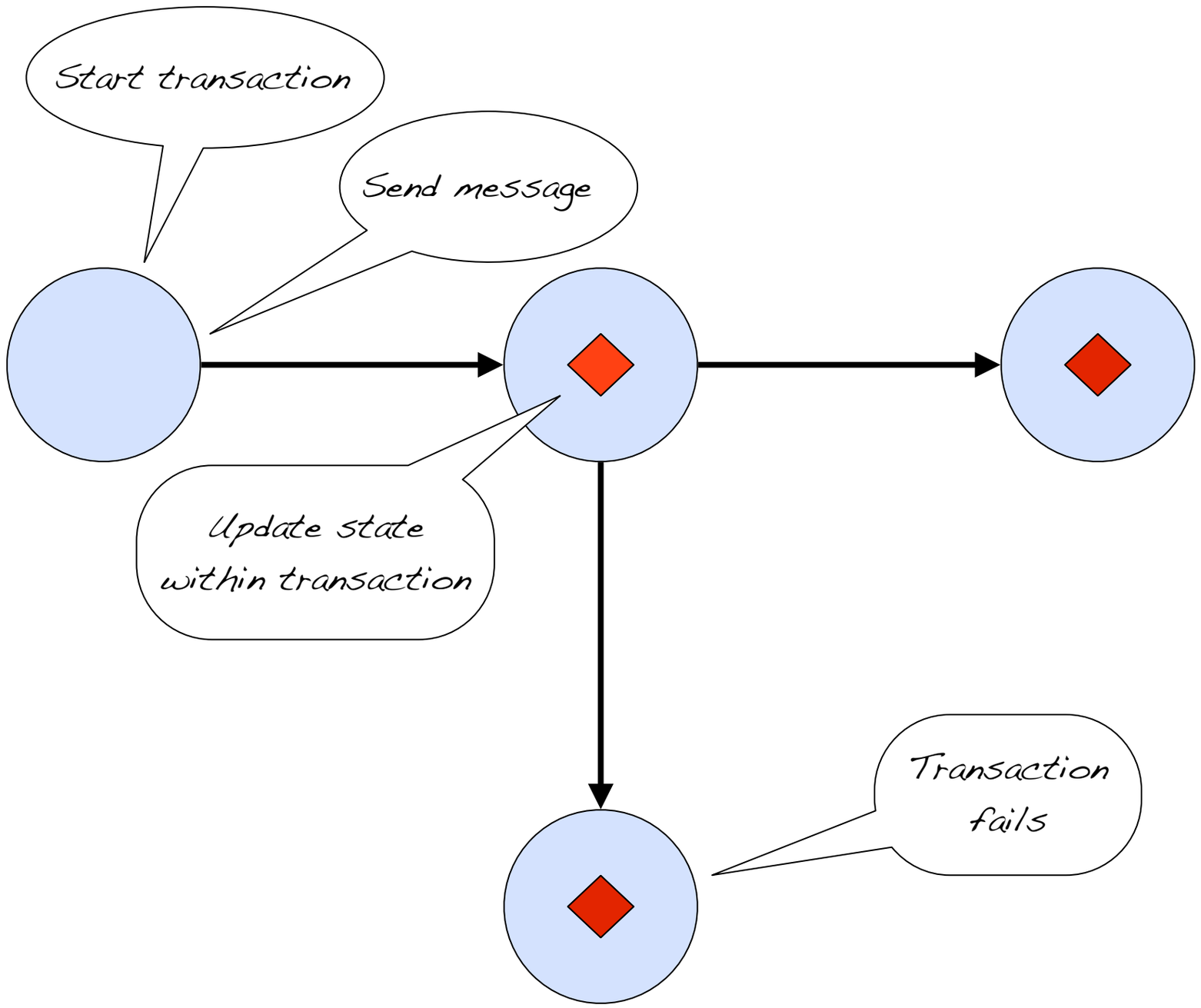
Transactors



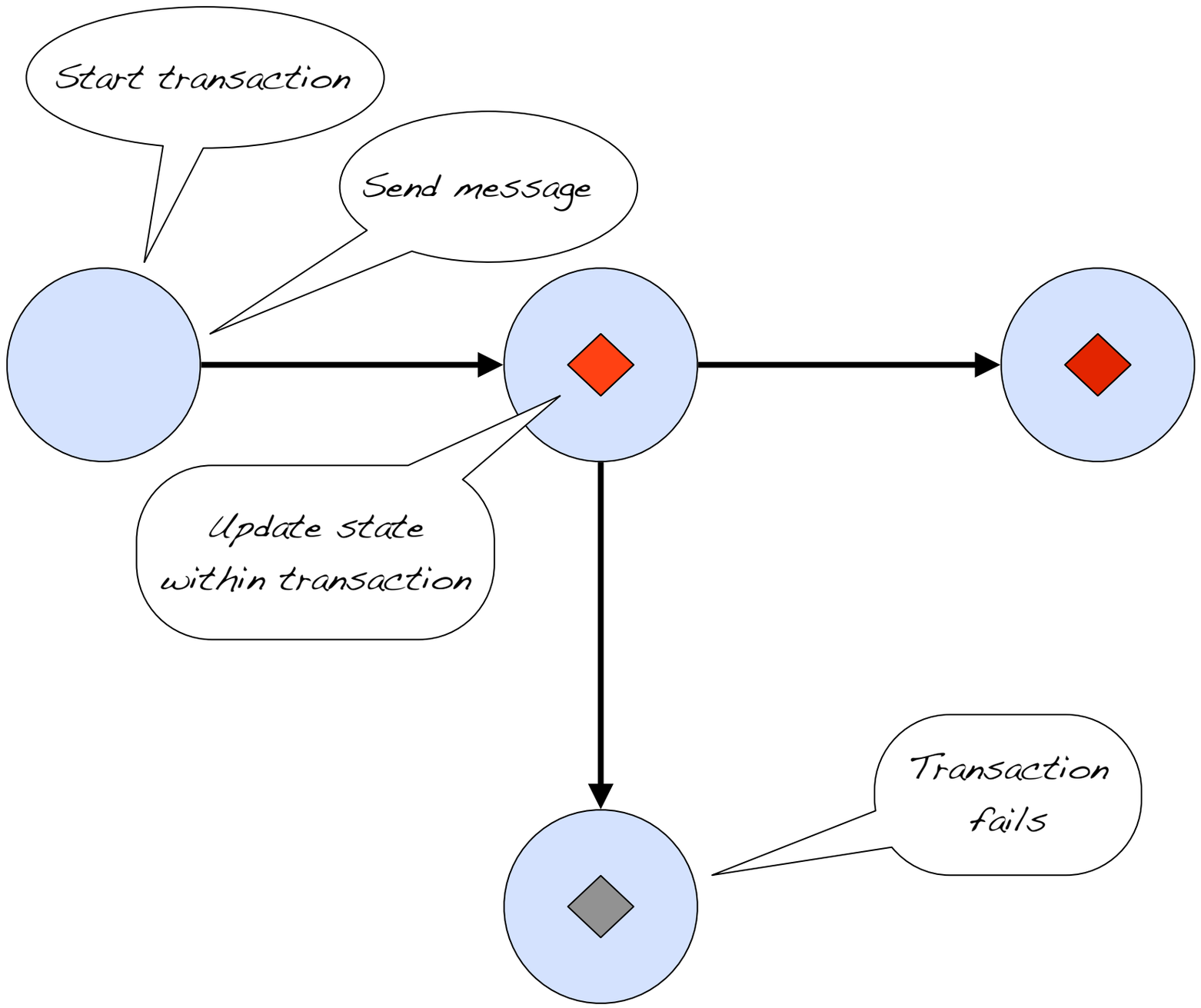
Transactors



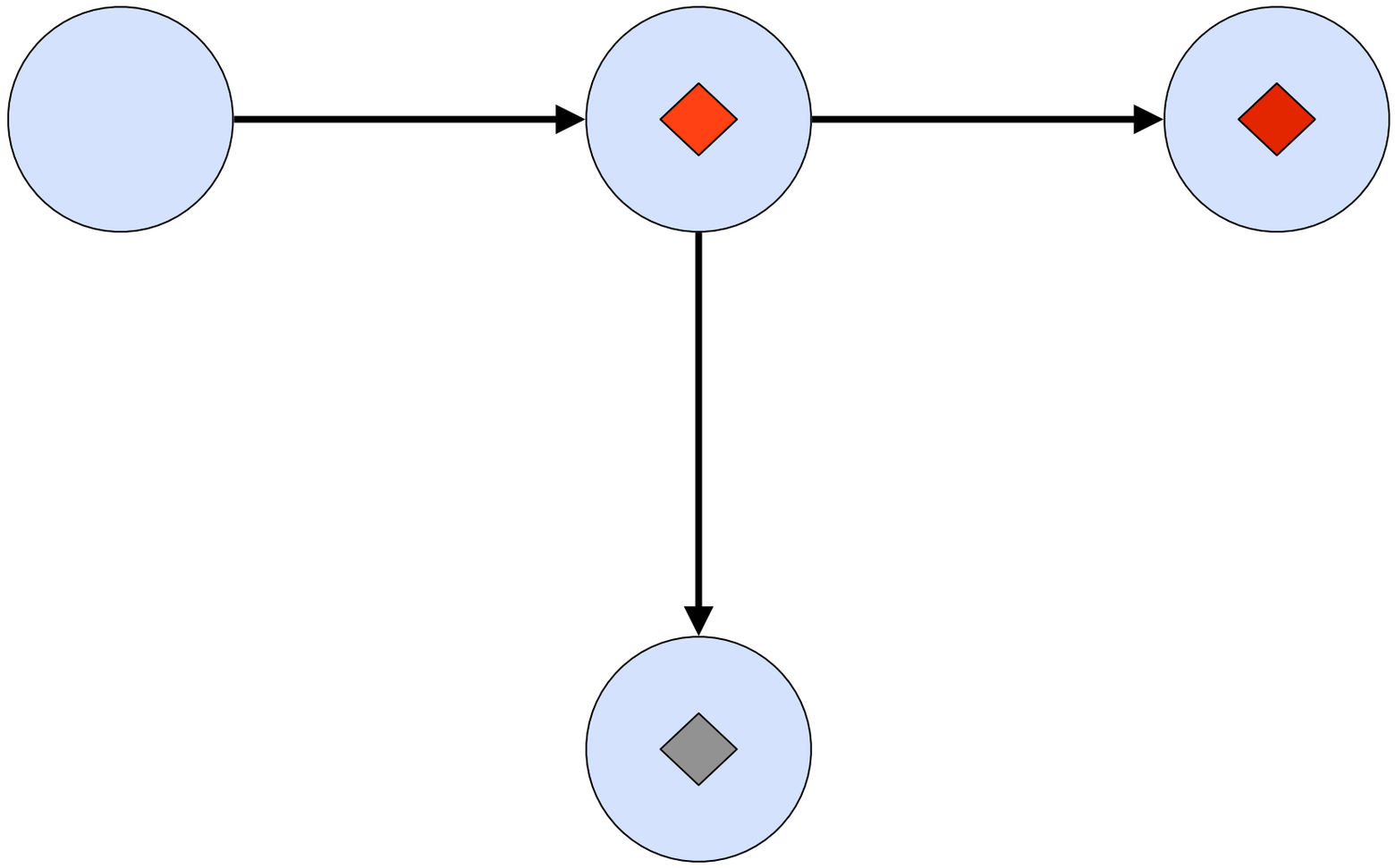
Transactors



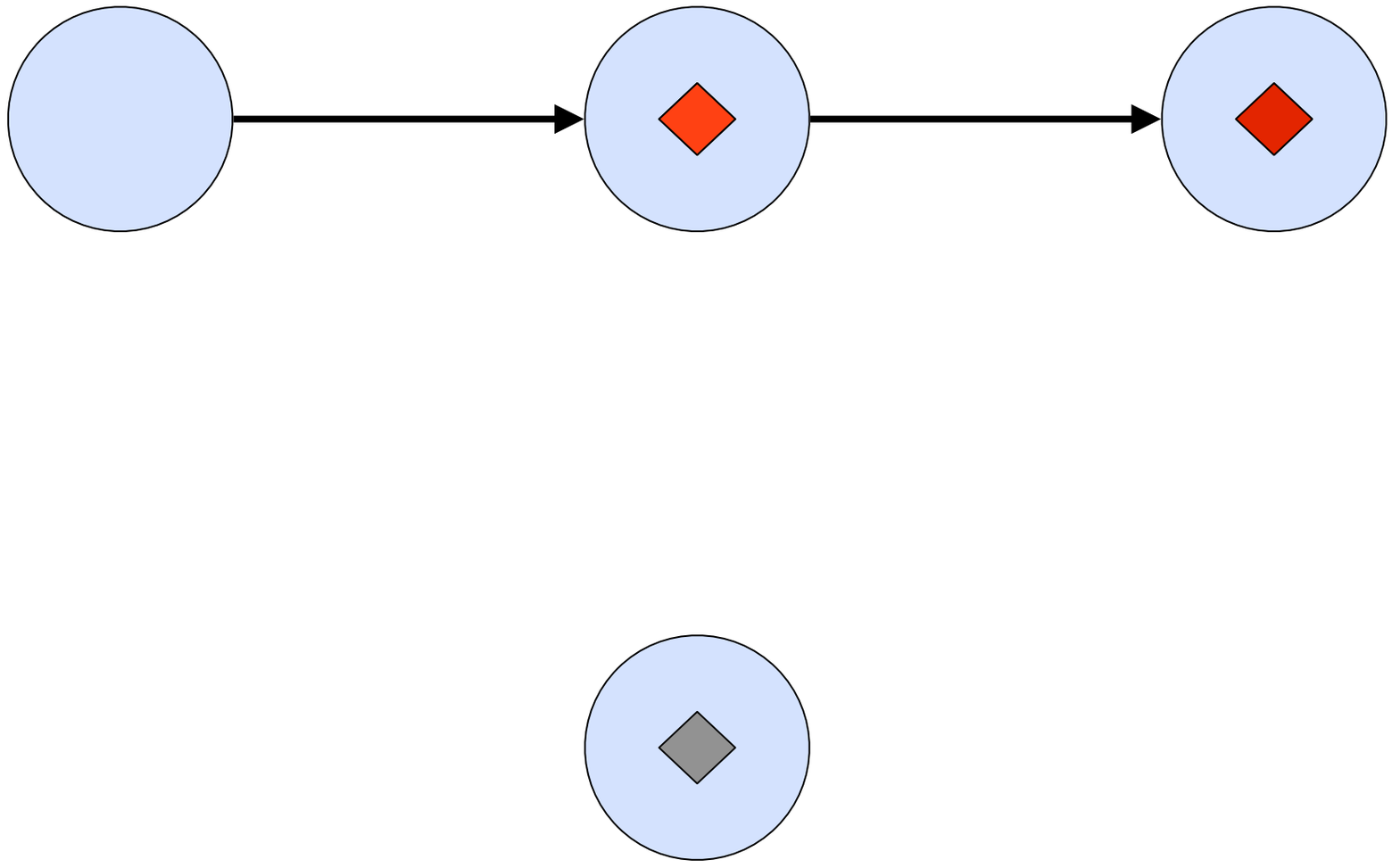
Transactors



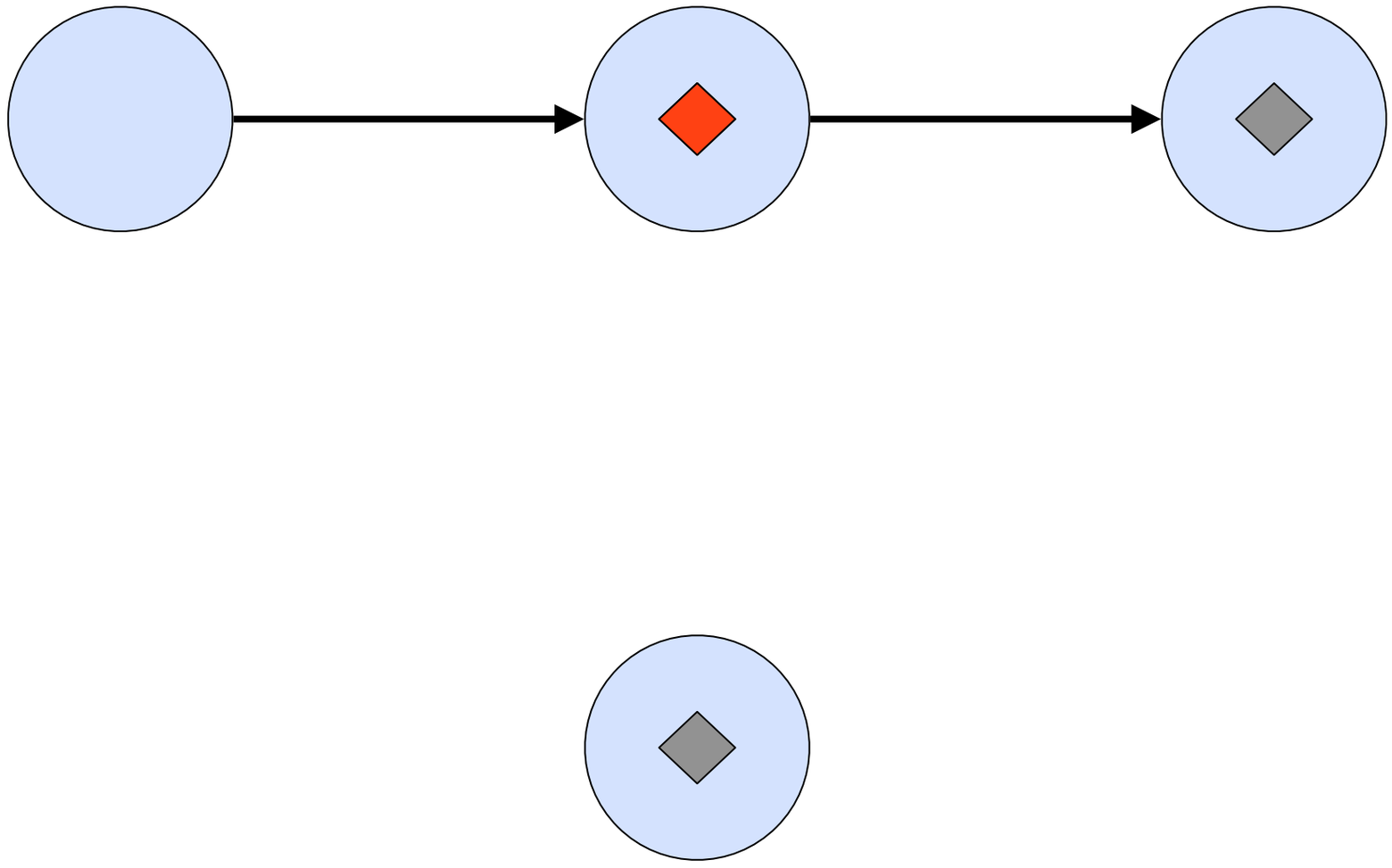
Transactors



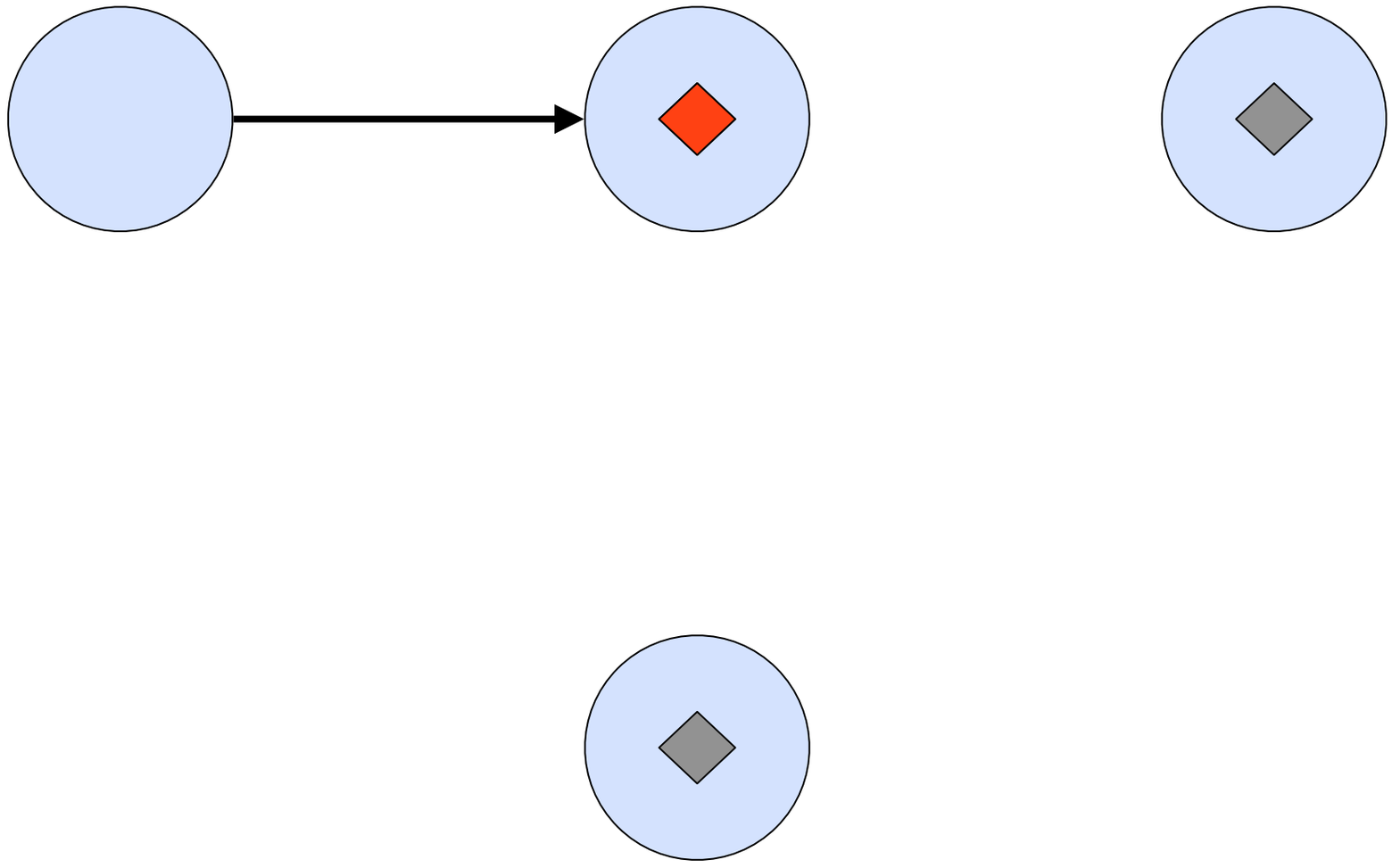
Transactors



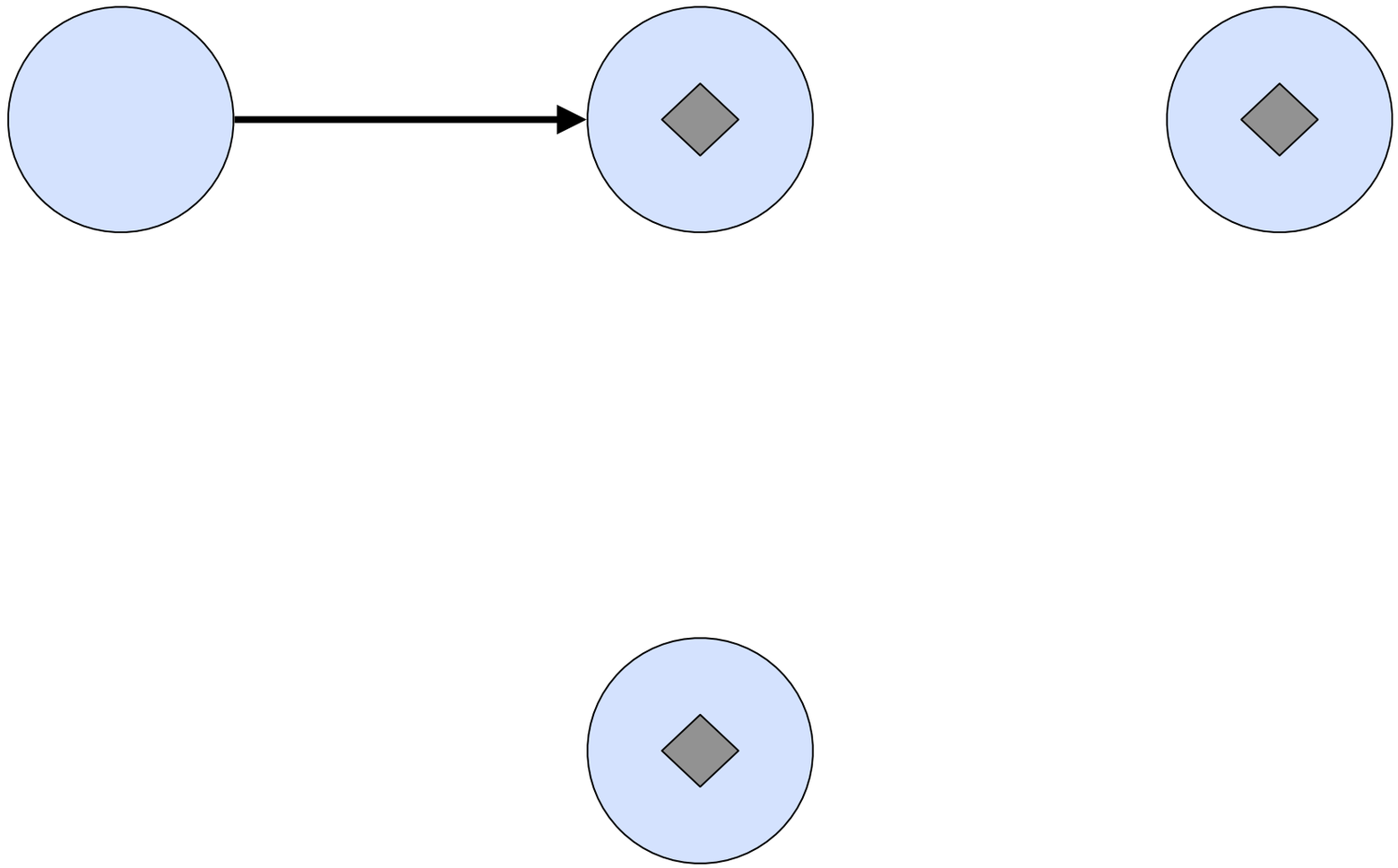
Transactors



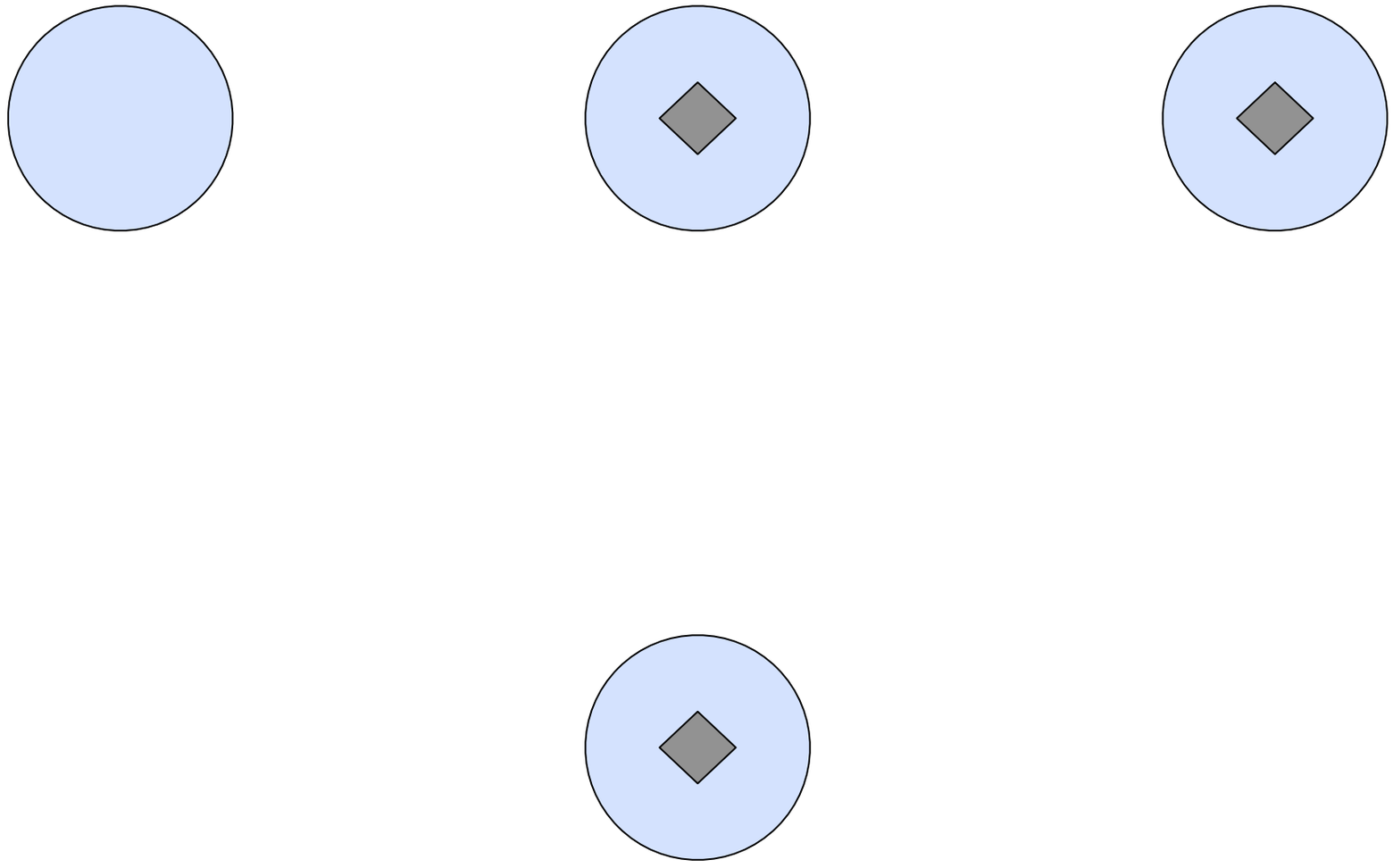
Transactors



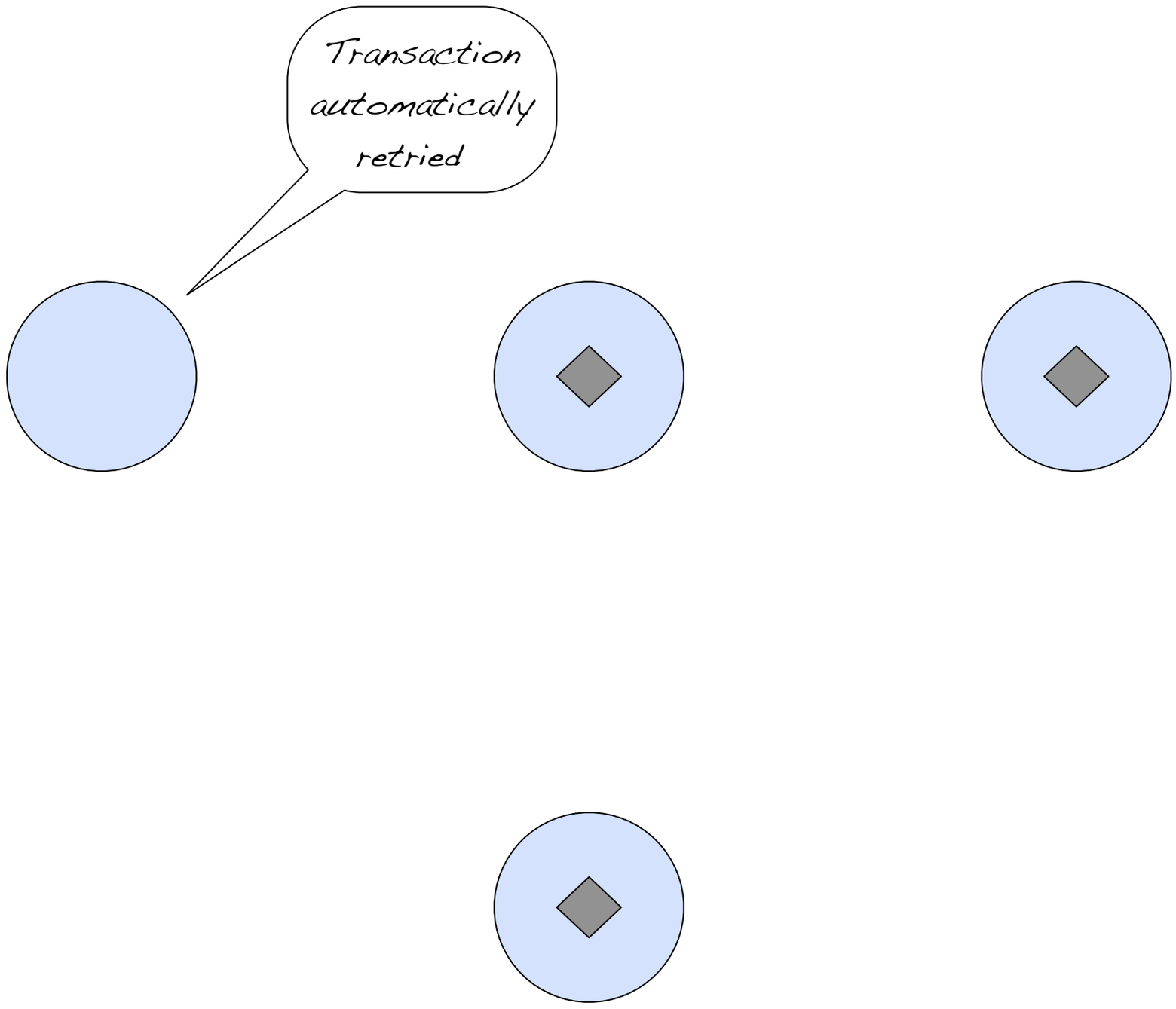
Transactors



Transactors



Transactors



STM: config

```
akka {
  stm {
    fair = on
    jta-aware = off
    timeout = 5
  }
}
```

STM: disable

```
TransactionManagement.disableTransactions
```

Akka Serialization

Serializers

Scala JSON

Java JSON

Protobuf

SBinary

Java

Serializers

Scala 2 JSON & JSON 2 Scala

```
val foo = new Foo
val json = Serializer.ScalaJSON.out(foo)
val fooCopy =
  Serializer.ScalaJSON.in(json).asInstanceOf[Foo]
```

Serializers

Scala 2 Binary & Binary 2 Scala

```
import sbinary.DefaultProtocol._
val users =
    ("user1", "passwd1") ::
    ("user2", "passwd2") ::
    ("user3", "passwd3") :: Nil
val bytes = Serializer.SBinary.out(users)

val usersCopy: List[Tuple2[String, String]]] =
  Serializer.SBinary.in(bytes)
```

Serializers

Protobuf

```
val pojo =
  ProtobufPOJO.getDefaultInstance.toBuilder
  .setId(1)
  .setName("protobuf")
  .setStatus(true).build
val bytes = pojo.toByteArray

val pojoCopy = Serializer.Protobuf.in(
  bytes, classOf[ProtobufPOJO])
```

Serializable

```
case class MyMessage(
  id: String,
  value: Tuple2[String, Int])
  extends Serializable.ScalaJSON

val message = MyMessage("id", ("hello", 34))
val json = message.toJSON
```

Modules

Akka Spring

Spring integration

```
<beans>
   <akka:typed-actor
      id="myActiveObject"
      interface="com.biz.MyPOJO"
      implementation="com.biz.MyPOJO"
      transactional="true"
      timeout="1000" />
   ...
</beans>
```

Spring integration

```
<akka:supervision id="my-supervisor">

  <akka:restart-strategy failover="AllForOne"
                         retries="3"
                         timerange="1000">
    <akka:trap-exits>
      <akka:trap-exit>java.io.IOException</akka:trap-exit>
    </akka:trap-exits>
  </akka:restart-strategy>

  <akka:typed-actors>
    <akka:typed-actor interface="com.biz.MyPOJO"
                      implementation="com.biz.MyPOJOImpl"
                      lifecycle="permanent"
                      timeout="1000">
    </akka:typed-actor>
  </akka:typed-actors>
</akka:supervision>
```

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"

  def receive = produce // use default impl
}
```

Camel: producer

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

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

Akka Persistence

STM gives us

Atomic

Consistent

Isolated

Persistence module turns STM into

Atomic

Consistent

Isolated

Durable

Akka Persistence API

```
// transactional Cassandra-backed Map
val map = CassandraStorage.newMap

// transactional Redis-backed Vector
val vector = RedisStorage.newVector

// transactional Mongo-backed Ref
val ref = MongoStorage.newRef
```

Get data by id

```
// transactional Cassandra-backed Map
val map = CassandraStorage.getMap(uuid)

// transactional Redis-backed Vector
val vector = RedisStorage.getVector(uuid)

// transactional Mongo-backed Ref
val ref = MongoStorage.getRef(uuid)
```

For Redis only (so far)

```
val queue: PersistentQueue[ElementType] =
  RedisStorage.newQueue

val set: PersistentSortedSet[ElementType] =
  RedisStorage.newSortedSet
```

Akka HotSwap

HotSwap

```
actor ! HotSwap(Some({
    // new body
    case Ping =>
        ...
    case Pong =>
        ...
}))
```

Akka AMQP

AMQP: producer

```
val producer = AMQP.newProducer(
   config,
   hostname, port,
   exchangeName,
   serializer,
    None, None, // listeners
   100)

producer ! Message("Hi there", routingId)
```

AMQP: consumer

```
val consumer = AMQP.newConsumer(
  config, hostname, port, exchangeName,
  ExchangeType.Direct, serializer,
  None, 100, passive, durable,
  Map[String, AnyRef())

consumer ! MessageConsumerListener(
  queueName, routingId, actor {
    case Message(payload, _, _, _, _) =>
      ... // process message
  })
```

How to run it?

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

Akka Kernel

Start Kernel

```
java -jar akka-0.9.1.jar \
   -Dakka.config=<path>/akka.conf
```

Or

```
export AKKA_HOME=<path to akka dist>
java -jar $AKKA_HOME/dist/akka-0.9.1.jar
```

Boot classes

```
class Boot {
  val supervisor = Supervisor(
    SupervisorConfig(
      RestartStrategy(OneForOne, 3, 100),
      Supervise(
        actorOf[Counter], LifeCycle(Permanent)) ::
      Supervise(
        actorOf[Chat], LifeCycle(Permanent)) ::
      Nil)))
}
```

Boot config

```
akka {
  boot = ["sample.rest.Boot",
          "sample.comet.Boot"]
  ...
}
```

Learn more

http://akkasource.org

EOF