

Let it crash:

using actors for fault-tolerance,
scalability & concurrency

Jonas Bonér
Scalable Solutions AB

Copyright 2010 - Scalable Solutions AB

Facts

- 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

Actors

one tool in the toolbox

Actors

- Originates in a 1973 paper by Carl Hewitt
- Implemented in Erlang, Occam, Oz
- Encapsulates state and behavior
- Closer to the definition of OO than classes

Alan Kay

(father of SmallTalk and OOP)

“OOP to me means only messaging, local retention and protection and hiding of state-process.”

“Actually I made up the term “object-oriented”, and I can tell you I did not have C++ in mind.”

Replace C++ with Java or C#

Actors

- Implements Message-Passing Concurrency
- Share **NOTHING**
- Isolated **lightweight** processes
- Communicates through **messages**
- **Asynchronous** and **non-blocking**

Actor Model of Concurrency

- **No shared state**
... hence, nothing to synchronize.
- Each actor has a **mailbox** (message queue)

Actor Model of Concurrency

- Non-blocking **send**
- “Blocking” **receive**
- Messages need to be **immutable**
- Highly **performant** and **scalable**
- Event-driven programming

Actor Model of Concurrency

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

Akka Actors

- Asynchronous
 1. Fire-and-forget
 2. Futures (Send Receive Reply Eventually)
- Message loop with pattern (message) matching
- Erlang-style

Two different models

- **Thread**-based
- **Event**-based
 - **Very** lightweight
 - Can easily create **millions** on a single workstation (6.5 million on 4 G RAM)

Actor libs for the JVM

- > Akka (Java/Scala)
- > scalaz actors (Scala)
- > Lift Actors (Scala)
- > Scala Actors (Scala)
- > Kilim (Java)
- > Jetlang (Java)
- > Actor's Guild (Java)
- > Actorom (Java)
- > FunctionalJava (Java)
- > GPars (Groovy)

Akka

Simpler Concurrency, Fault tolerance & Scalability

What **is** Akka?

STM

Actors

Dataflow

Distributed

Secure

Persistent

RESTful

Comet

Actors

```
case object Tick

class Counter extends Actor {
  private var counter = 0

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

Actors

anonymous

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

Actors

anonymous

```
val worker = Actor.init {  
    ... // init  
} receive {  
    case Work(fn) => fn()  
}
```

Send: !

```
// fire-forget  
counter ! Tick
```

Send: !!

```
// uses Future with default timeout
val resultOption = actor !! Message

val result =
    resultOption getOrElse defaultResult
```

Send: !!!

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

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

Reply

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

Reply

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

Start / Stop

```
actor.start  
actor.stop
```

```
// many callbacks  
override def shutdown = {  
    ... // clean up before shutdown  
}
```

Active Objects: Java

```
public class Counter {  
    private int counter = 0;  
    public void count() {  
        counter++;  
        System.out.println(counter);  
    }  
}
```

Create: POJO

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

Create:

Interface & Implementation

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

Create: POSO

```
val counter = ActiveObject.newInstance(  
  classOf[Counter], 1000)
```

Send message

`counter.count`

Good **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)

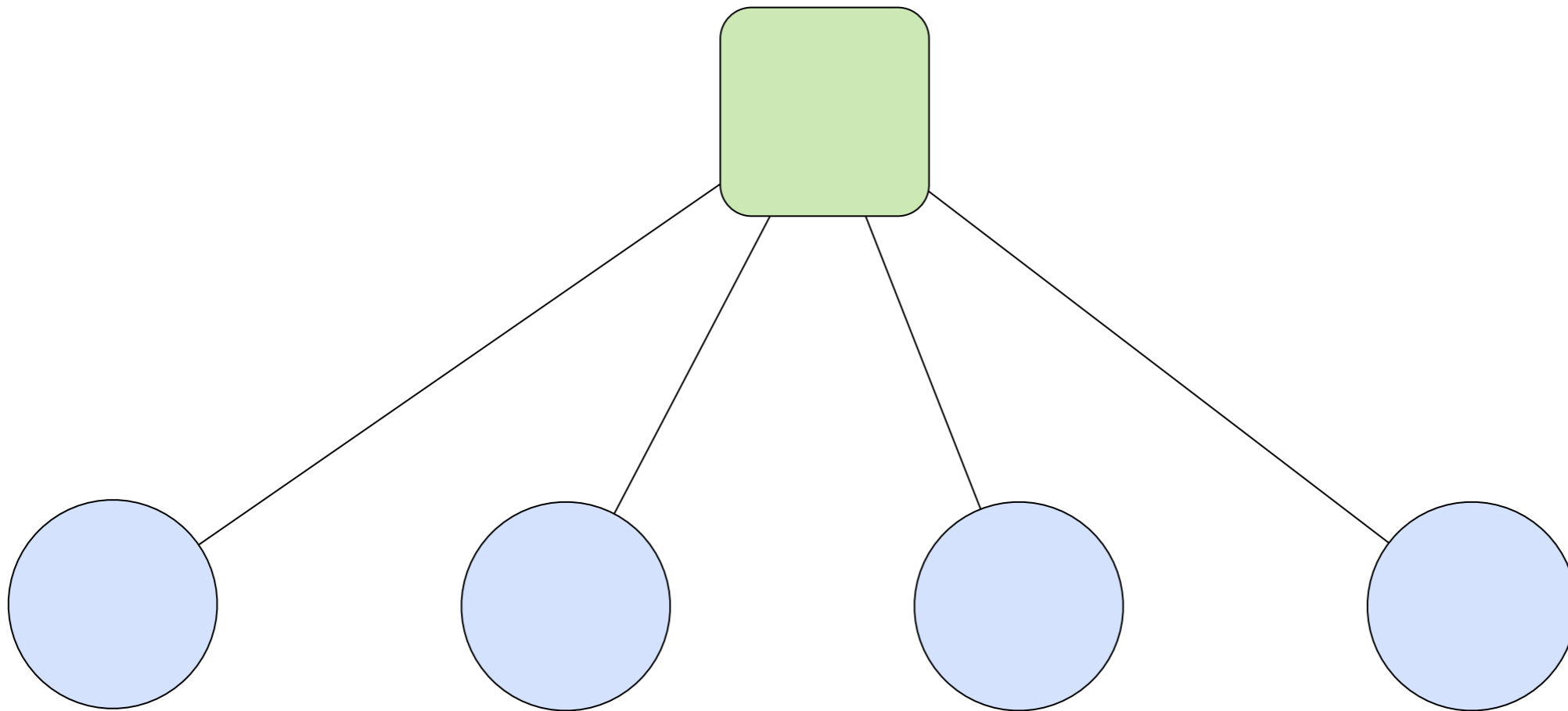
// lists
actor ! List("bill", "bob", "alice")
```

Let it crash
fault-tolerance

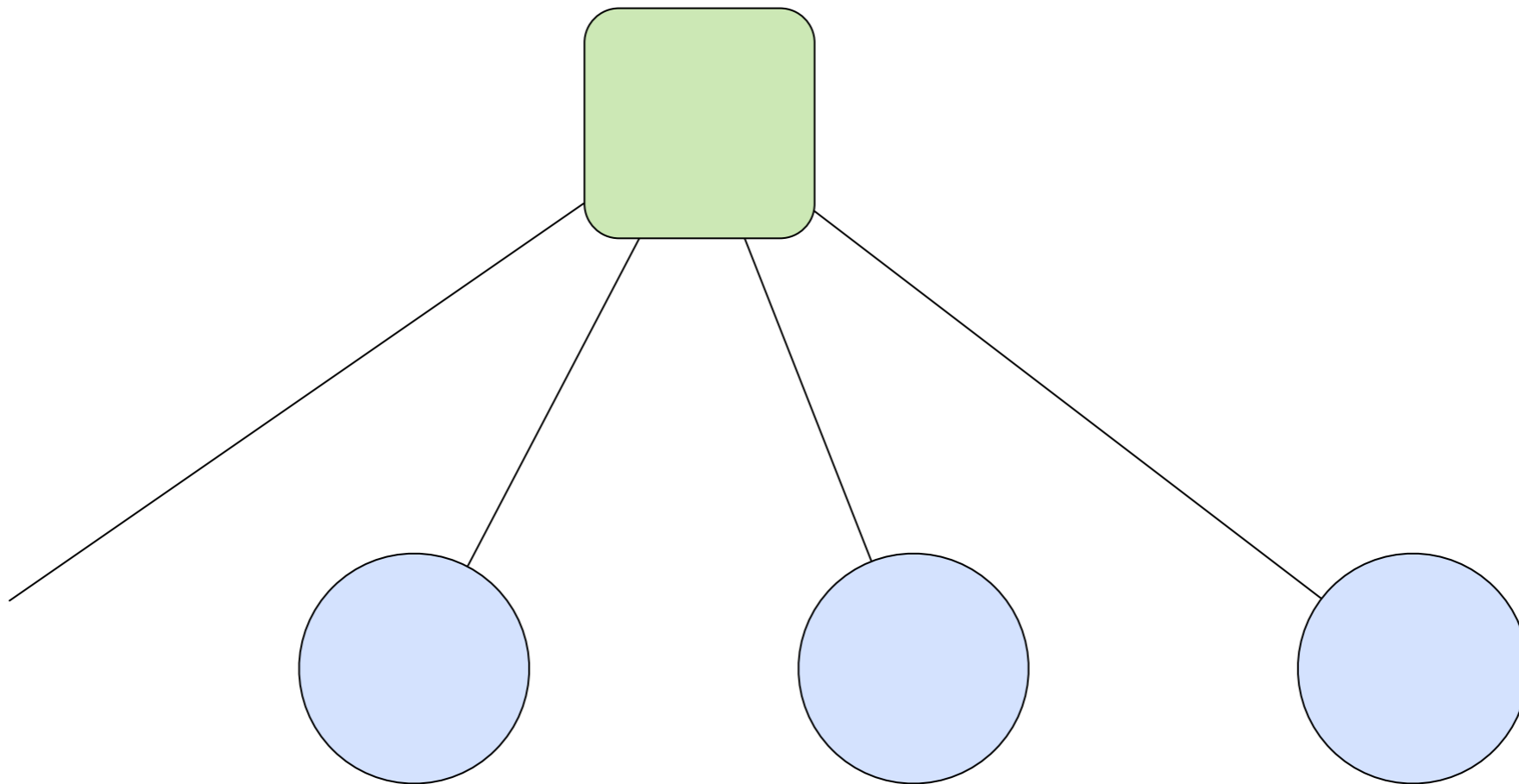
Stolen from
Erlang

9 nines

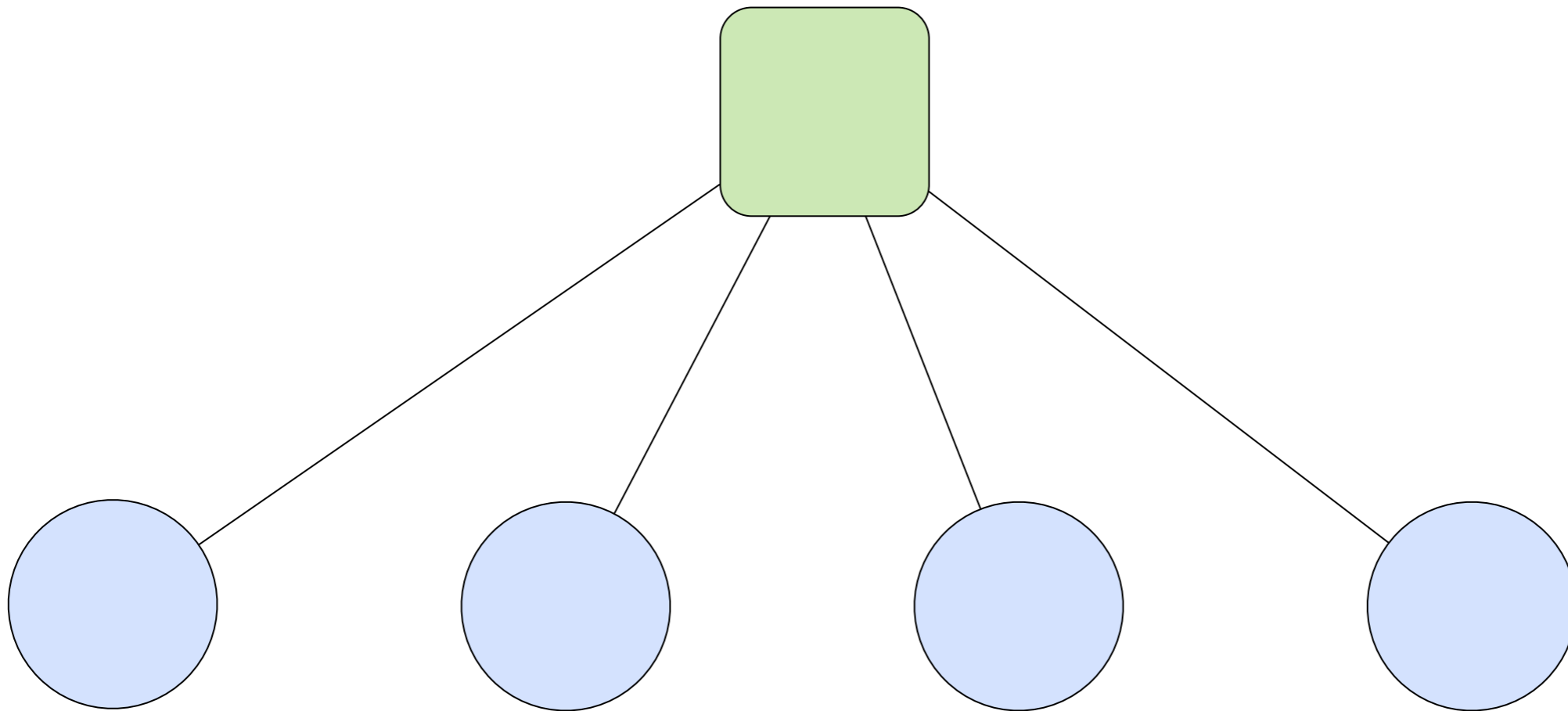
OneForOne



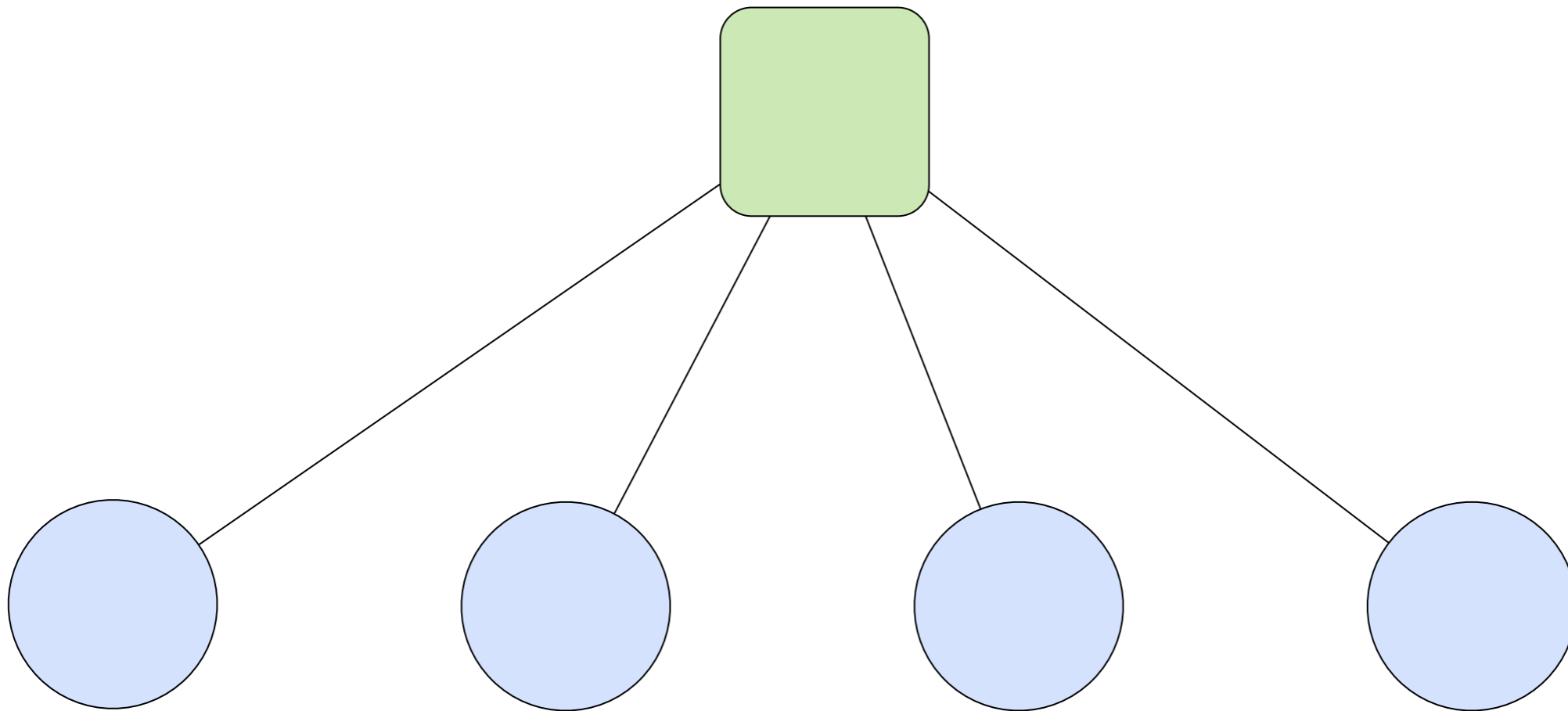
OneForOne



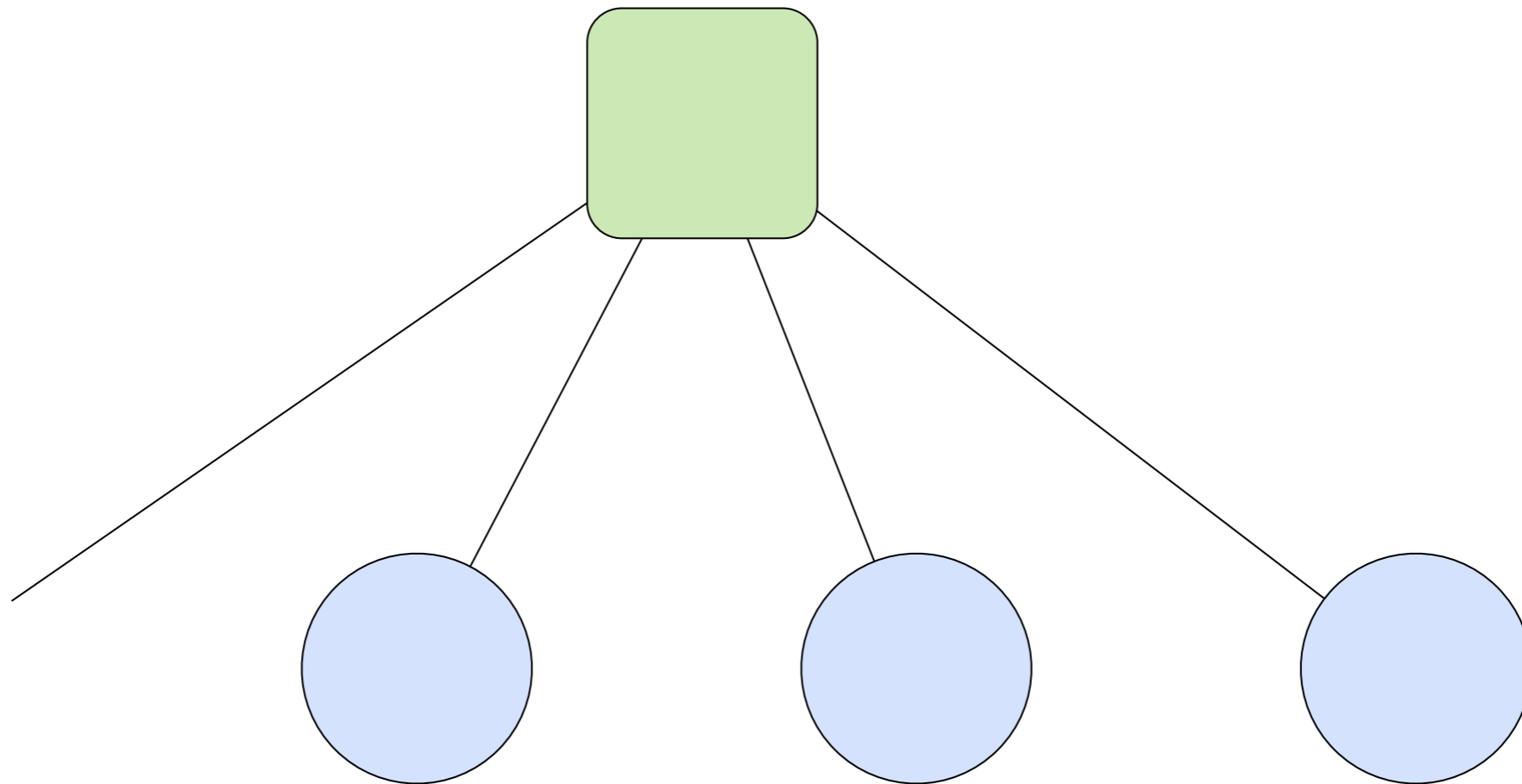
OneForOne



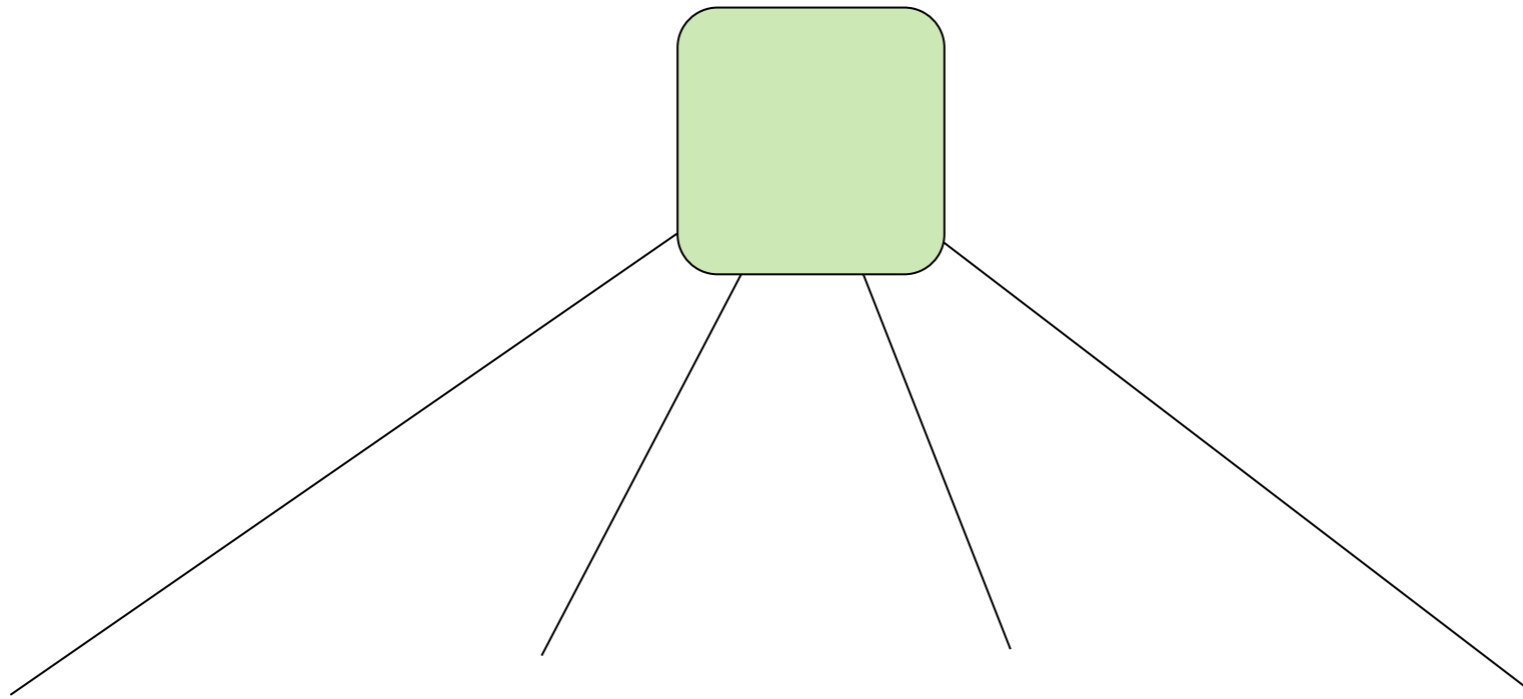
AllForOne



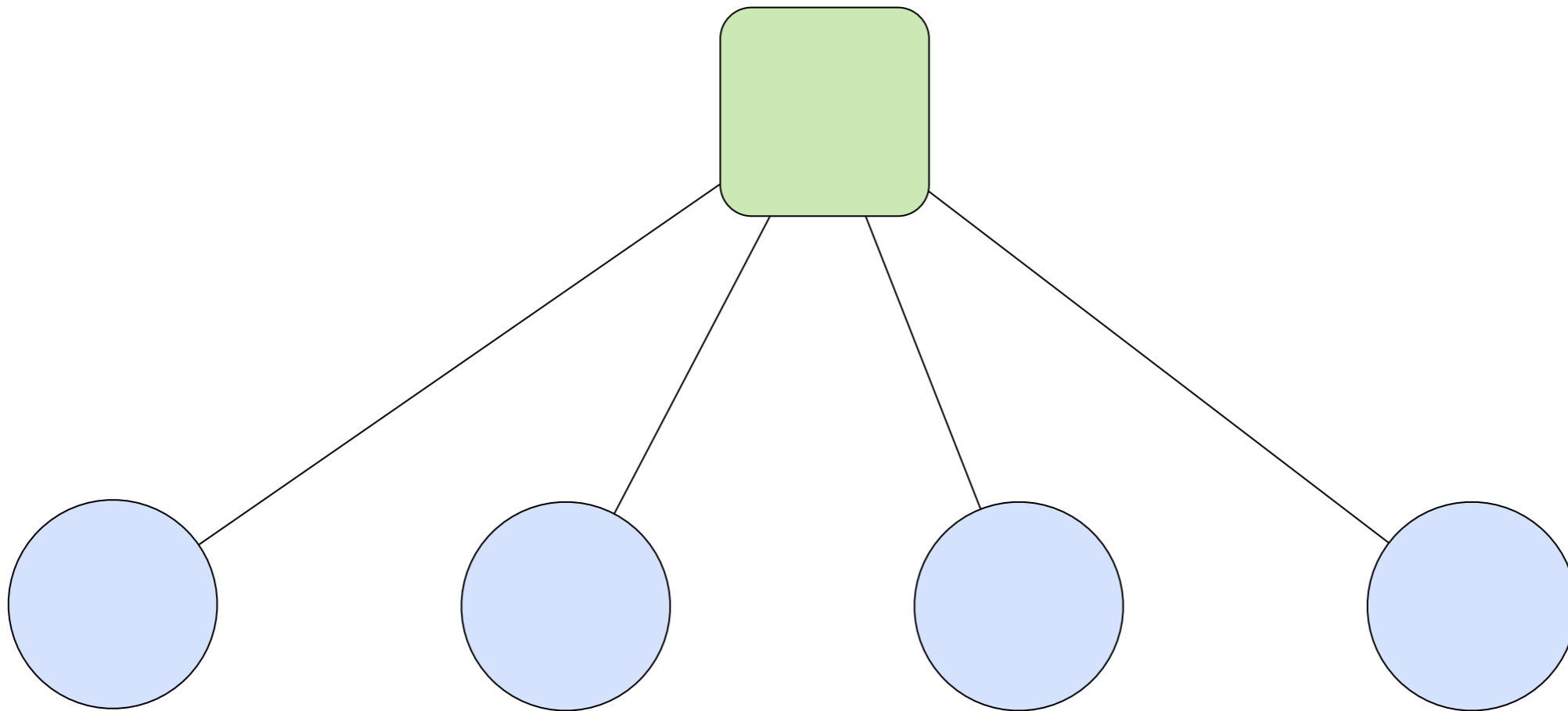
AllForOne



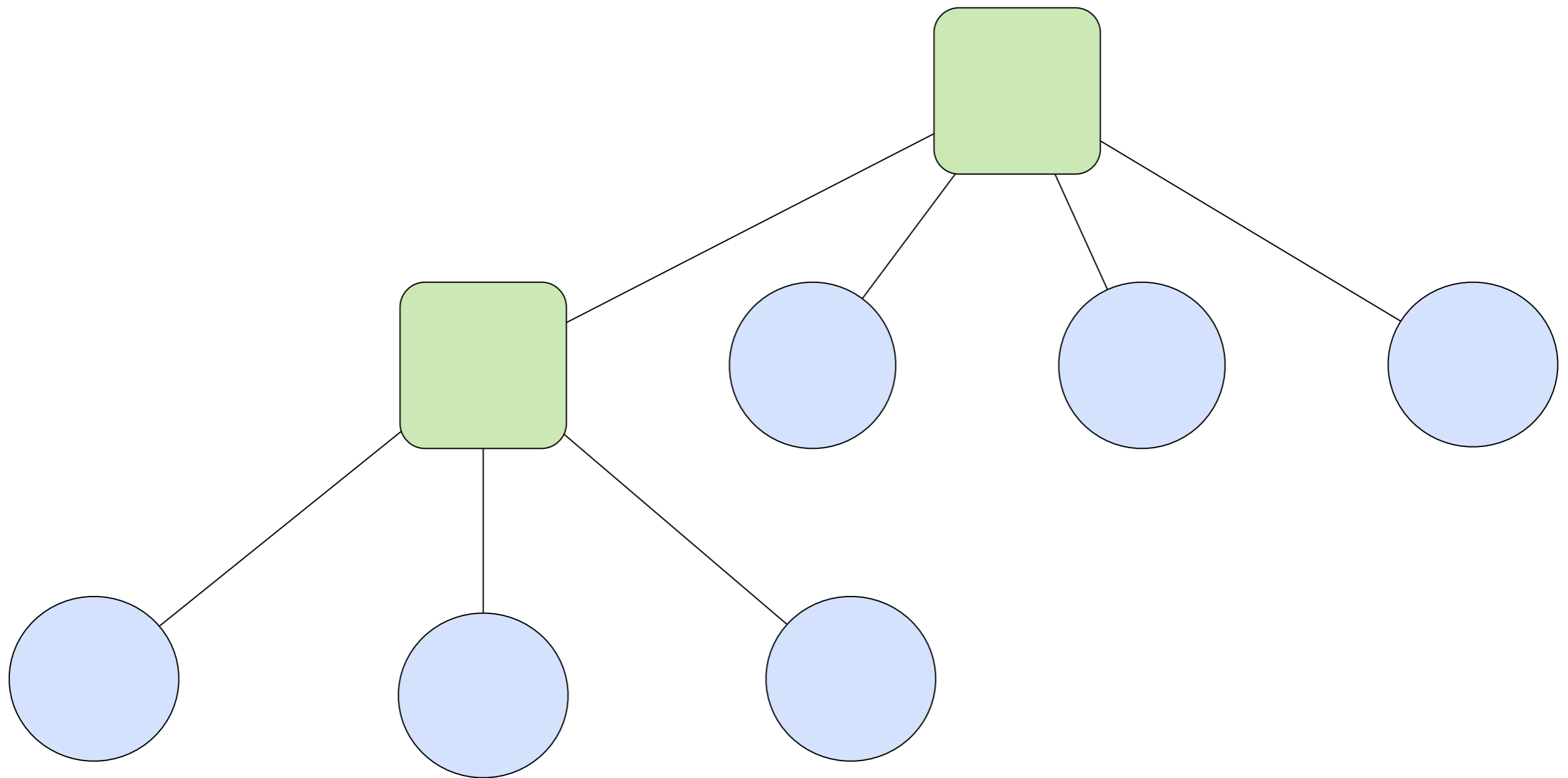
AllForOne



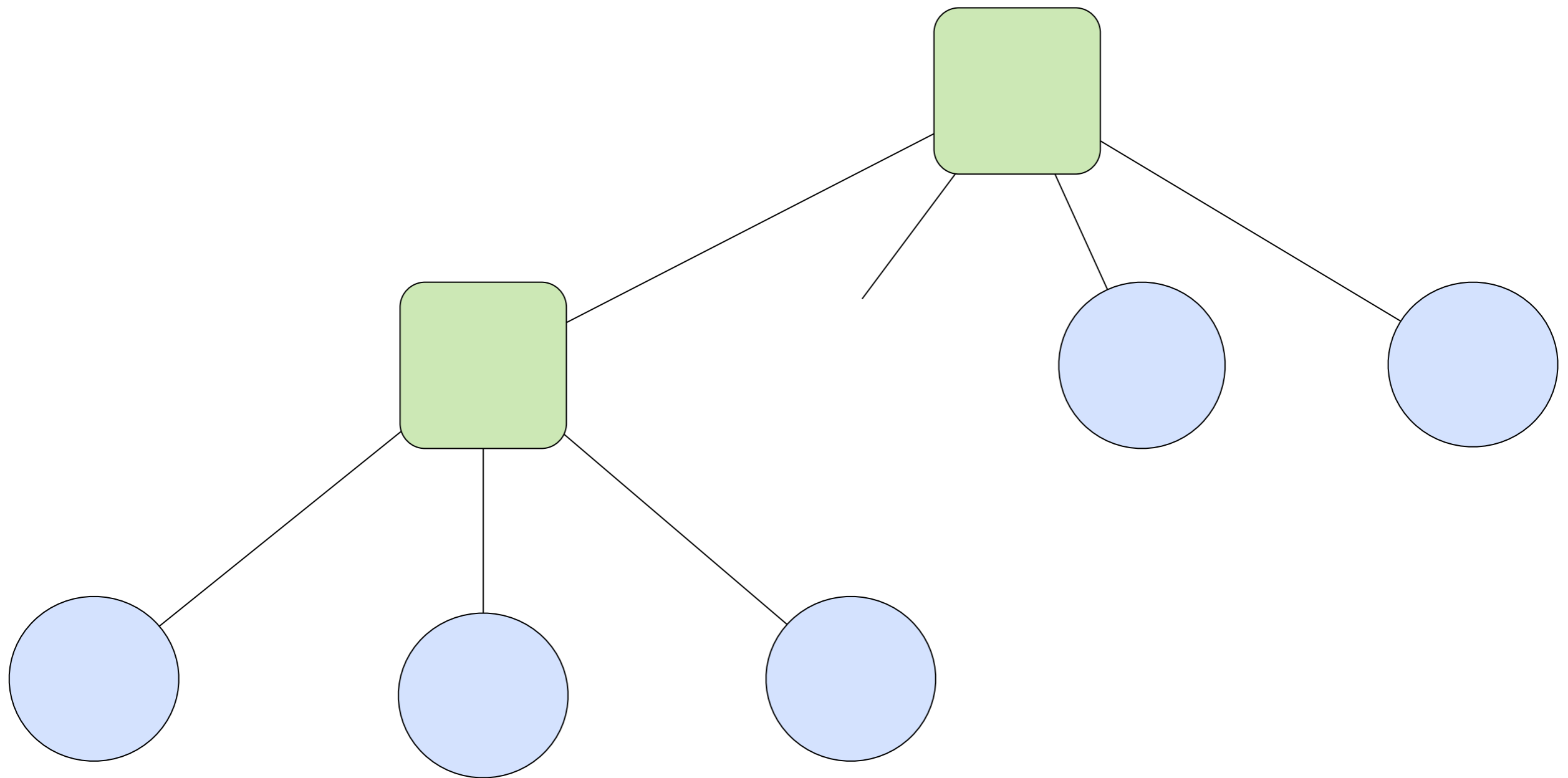
AllForOne



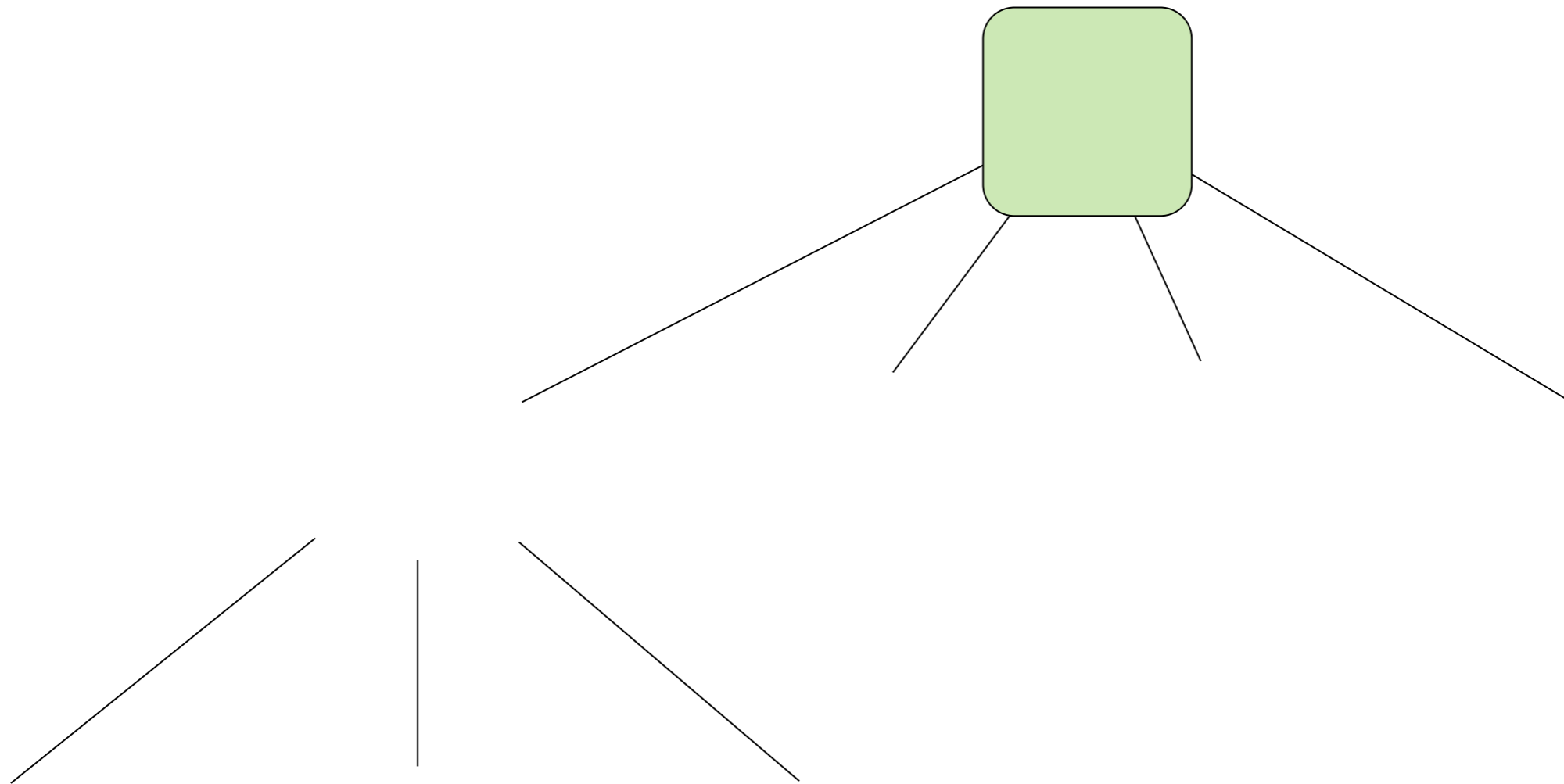
Supervisor hierarchies



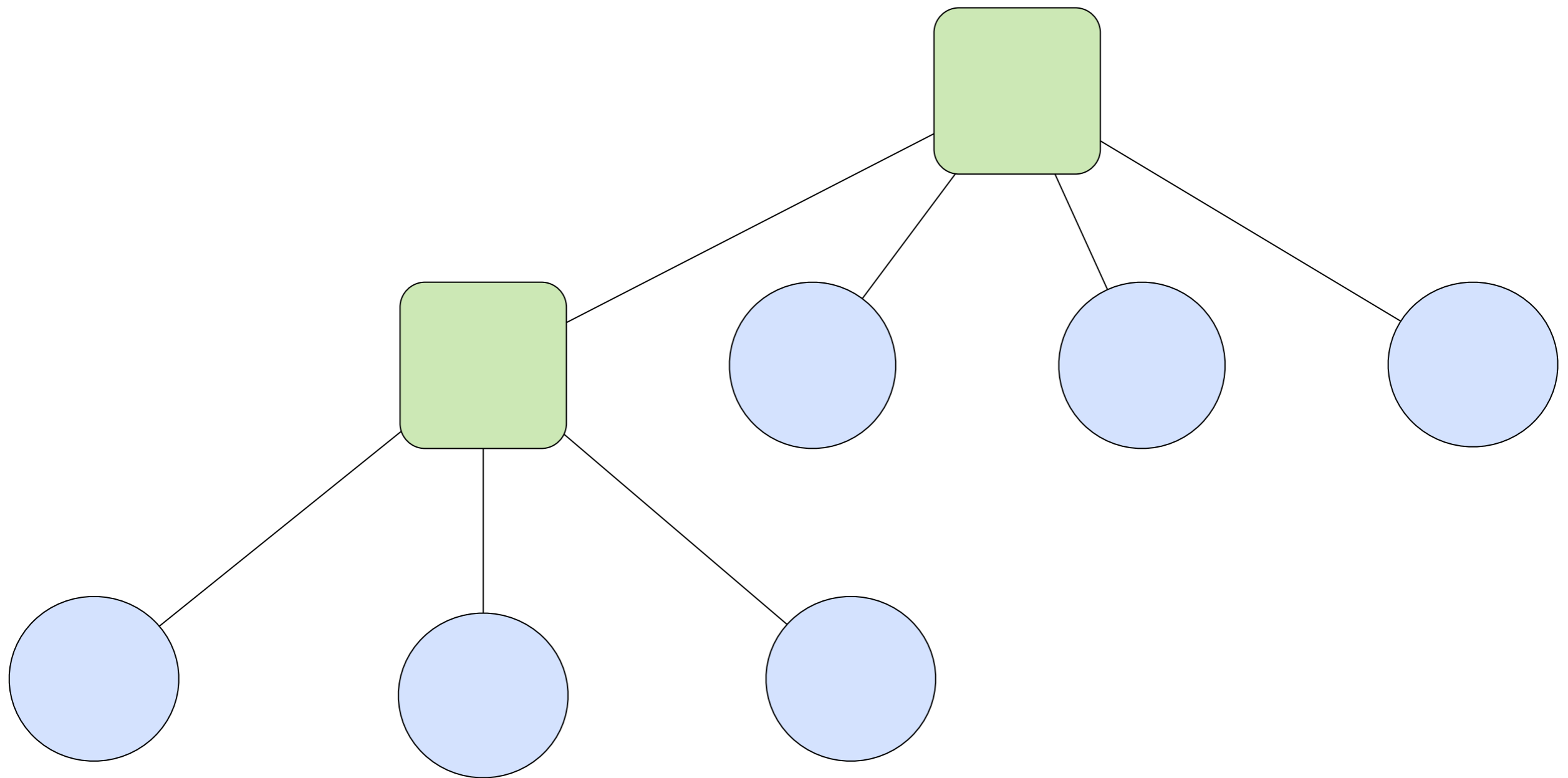
Supervisor hierarchies



Supervisor hierarchies



Supervisor hierarchies



Fault handlers

```
AllForOneStrategy(  
    maxNrOfRetries,  
    withinTimeRange)
```

```
OneForOneStrategy(  
    maxNrOfRetries,  
    withinTimeRange)
```

Linking

```
link(actor)
```

```
unlink(actor)
```

```
startLink(actor)
```

```
spawnLink(classOf[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 FaultTolerant extends Actor {  
  ...  
  override def preRestart(reason: Throwable) = {  
    ... // clean up before restart  
  }  
  override def postRestart(reason: Throwable) = {  
    ... // init after restart  
  }  
}
```

Declarative config

```
RestartStrategy(  
  AllForOne,    // restart policy  
  10,          // max # of restart retries  
  5000)        // within time in millis  
  
Lifecycle(  
  // Permanent: always be restarted  
  // Temporary: restarted if exited through ERR  
  Permanent)
```

Declarative config

```
object factory extends SupervisorFactory(  
  SupervisorConfig(  
    RestartStrategy(AllForOne, 3, 10000),  
    Supervise(  
      actor1,  
      Lifecycle(Permanent)) ::  
    Supervise(  
      actor2,  
      Lifecycle(Temporary)) ::  
    Nil))  
  
factory.newSupervisor.start
```

ActorRegistry

```
val actors =  
    ActorRegistry.actorsFor(FQN)
```

```
val actors =  
    ActorRegistry.actorsFor(classOf[...])
```

Remote Actors

Remote Server

```
// use host & port in config
RemoteNode.start
RemoteNode.start(classLoader)

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

Two types of remote actors

- **Client**-initiated and managed
- **Server**-initiated and managed

Client-managed

supervision works across nodes

```
// methods in Actor class
```

```
spawnRemote(classOf[MyActor], host, port)
```

```
startLinkRemote(actor, host, port)
```

```
spawnLinkRemote(classOf[MyActor], host, port)
```

Client-managed

moves actor to server

client manages through proxy

```
val actorProxy = spawnLinkRemote(  
  classOf[MyActor],  
  "darkstar",  
  9999)  
  
actorProxy ! Message(..)  
  
actorProxy.isInstanceOf[MyActor] // ==> true
```

Server-managed

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

```
RemoteNode.register(“service:id”, new MyService)
```

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

```
handle ! Message(..)
```

```
handle.isInstanceOf[MyActor] // ==> false
```

Cluster Membership

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

STMM

another tool in the toolbox

Software Transactional Memory (STM)

What is 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**

Akka STM

is based on the ideas of

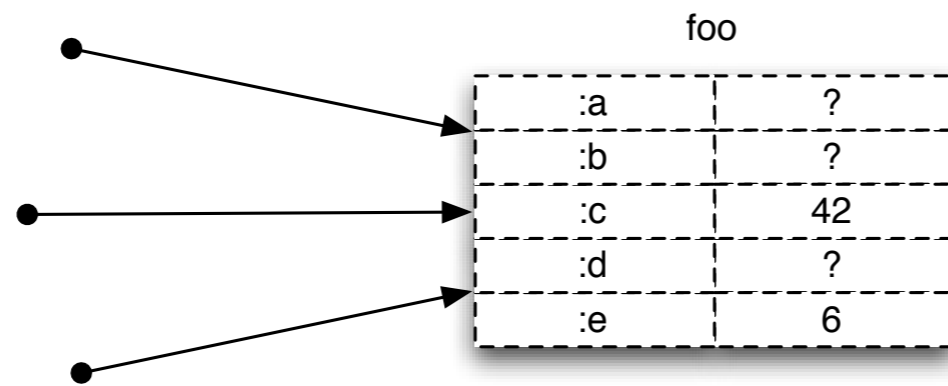
Clojure STM

Akka STM

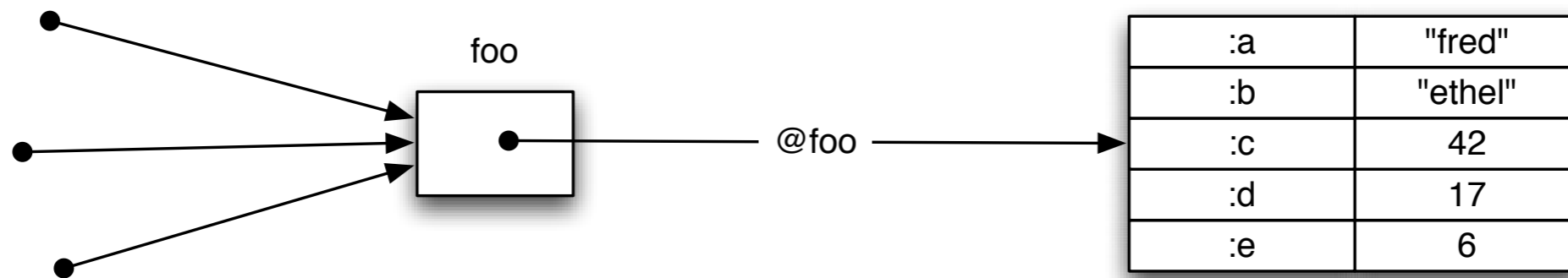
- Transactional Memory
 - Atomic, Consistent, Isolated (ACI)
 - MVCC
 - **Rolls back** in memory and **retries automatically** on clash
- Builds upon Multiverse

Managed References

- **Typical OO**: direct access to mutable objects



- **Managed Reference**: separates Identity & Value



Copyright Rich Hickey 2009

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 = TransactionalRef(Map[String, User]())  
  
val users = ref.get  
val newUsers =  
    users + ("bill" -> new User("bill", "secret"))  
  
ref.swap(newUsers)
```

Transactional datastructures

```
// wraps a Ref with a HashTrie  
val users = TransactionalState.newMap[String, User]  
  
// wraps a Ref with a Vector  
val users = TransactionalState.newVector[User]
```

STM: declarative Java API

```
@transactionrequired  
class UserRegistry {  
  
}
```

STM:API

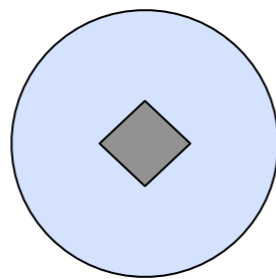
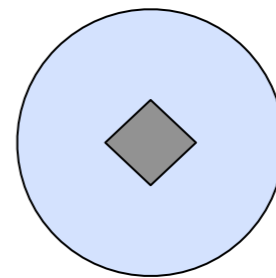
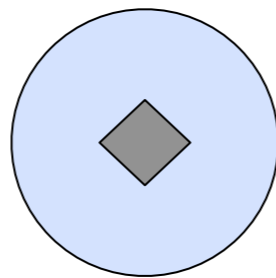
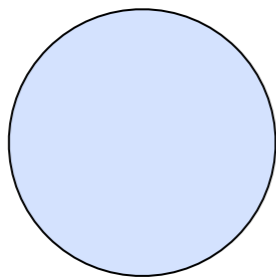
```
import se.scalablesolutions.akka.stm.Transaction._  
  
atomic {  
  .. // do something within a transaction  
}
```

Actors + STM =
Transactors

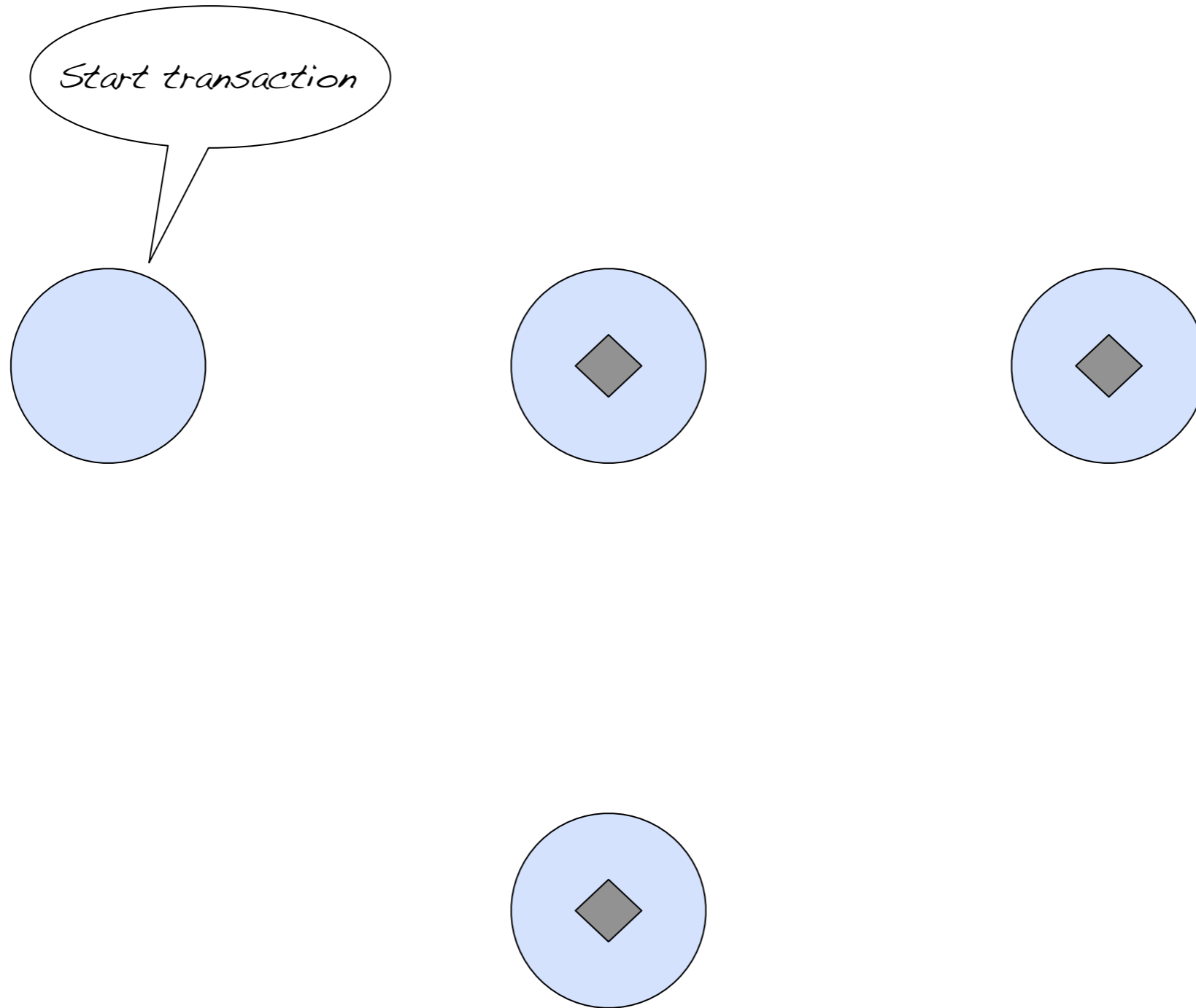
Transactors

```
class UserRegistry extends Transactor {  
  private lazy val storage =  
    TransactionalState.newMap[String, User]  
  
  def receive = {  
    case NewUser(user) =>  
      storage + (user.name -> user)  
    ...  
  }  
}
```

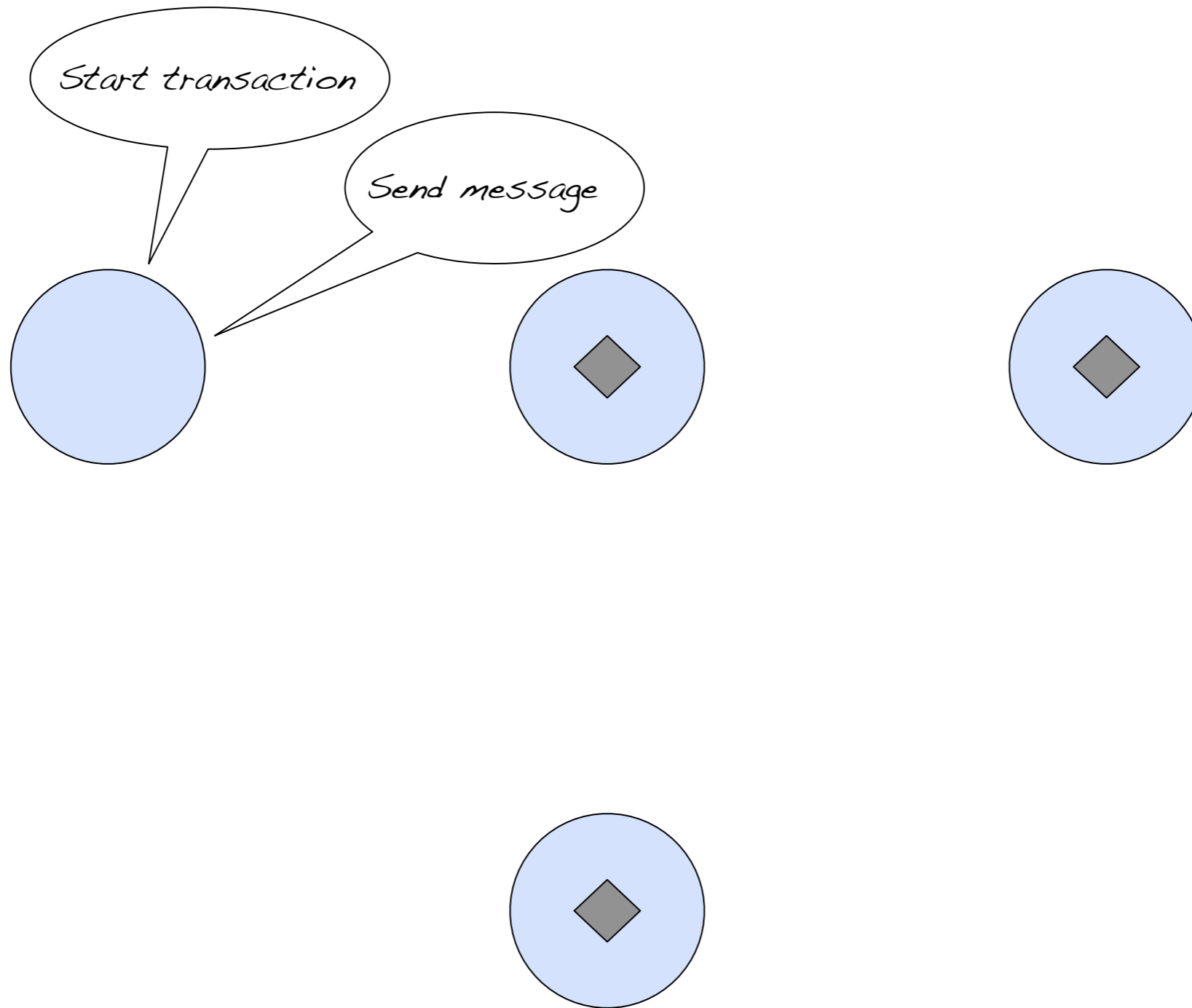
Transactors



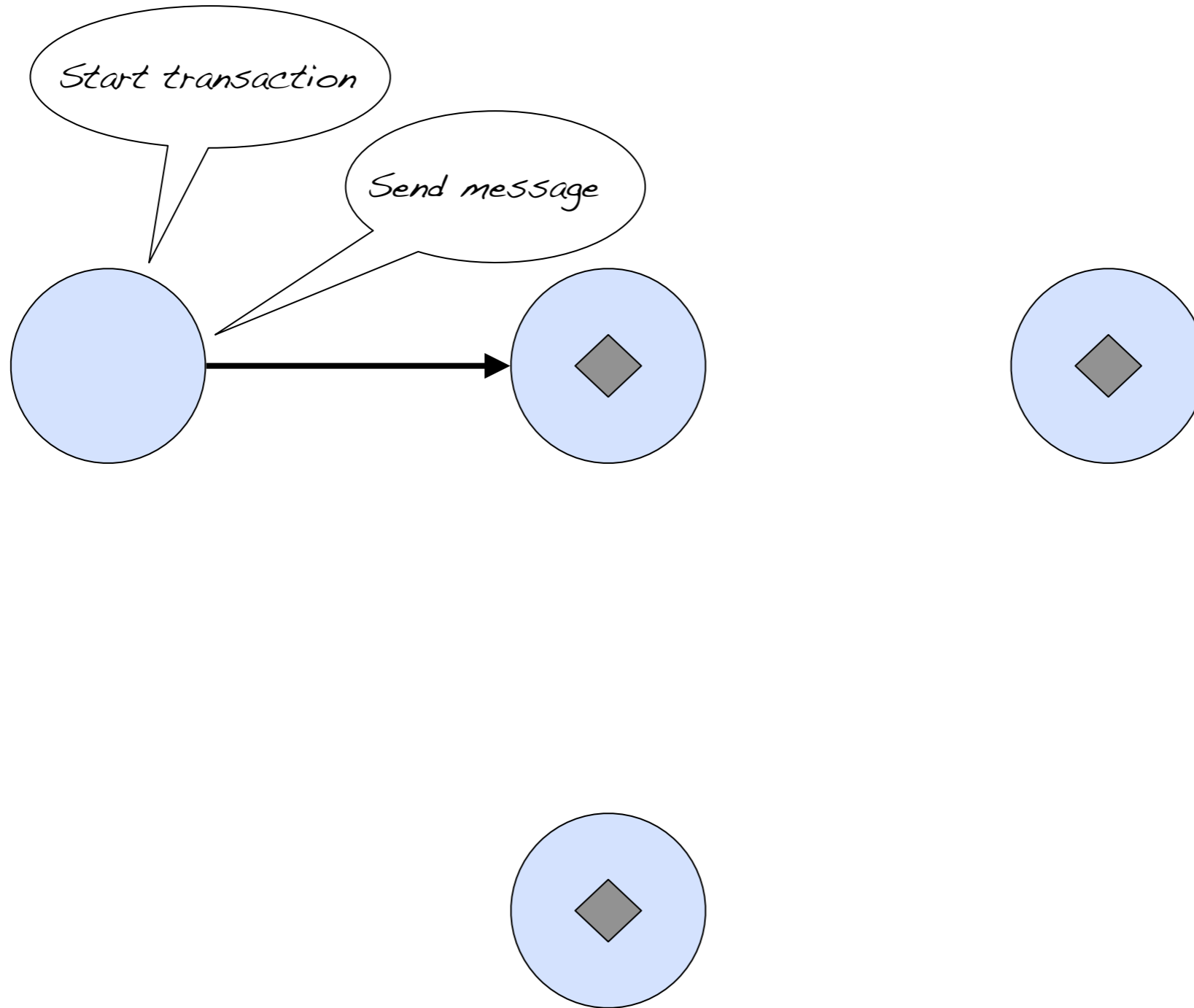
Transactors



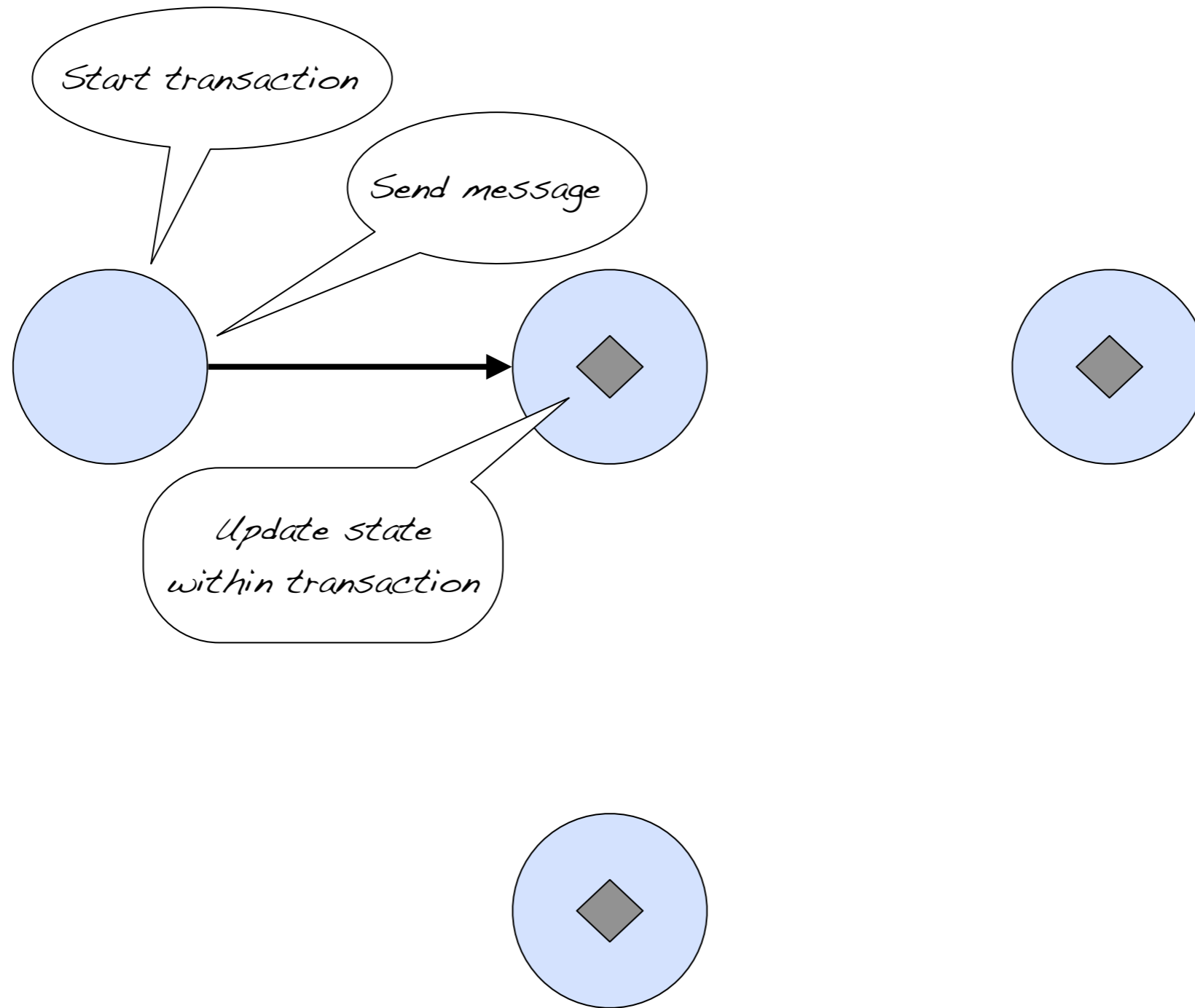
Transactors



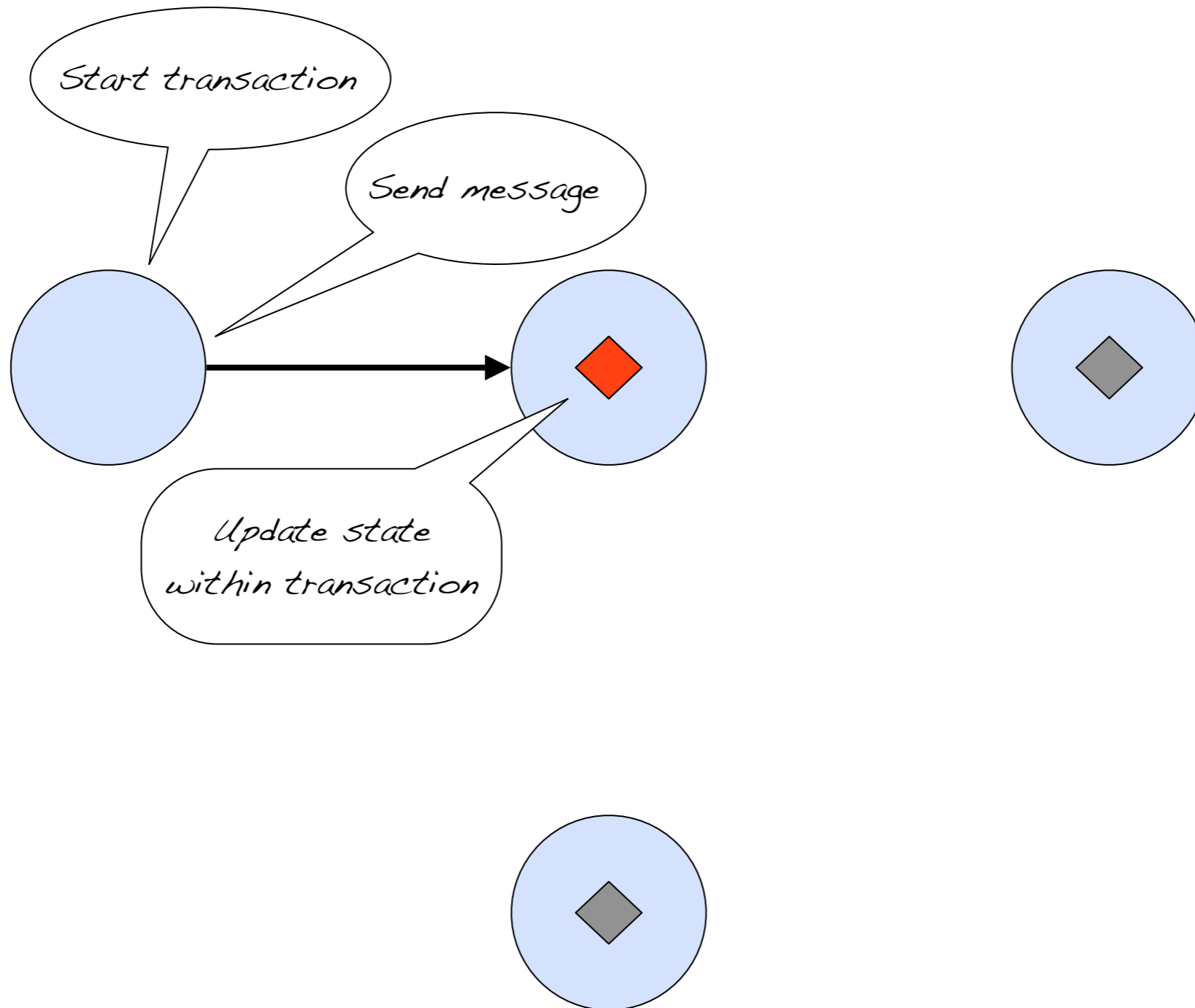
Transactors



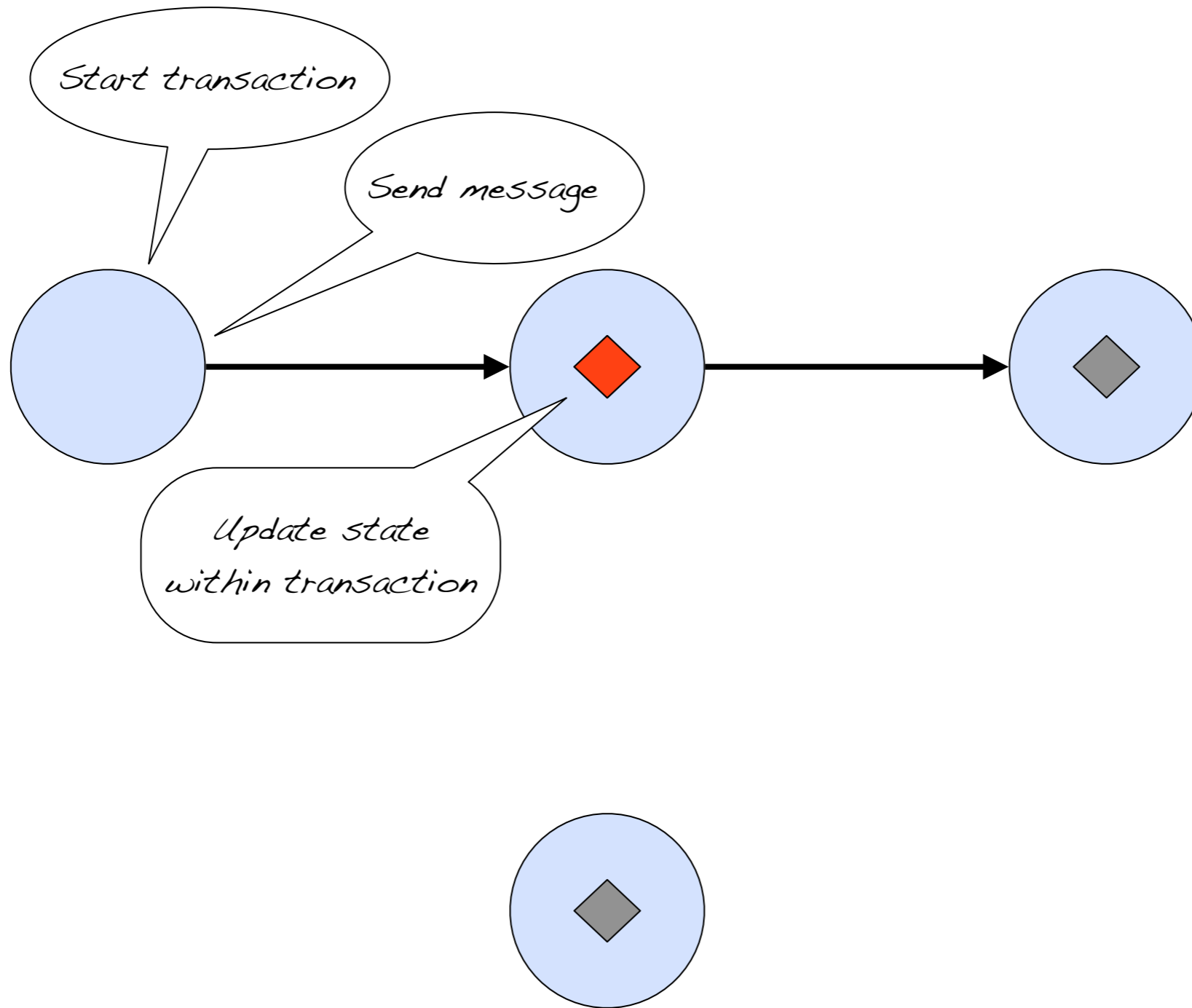
Transactors



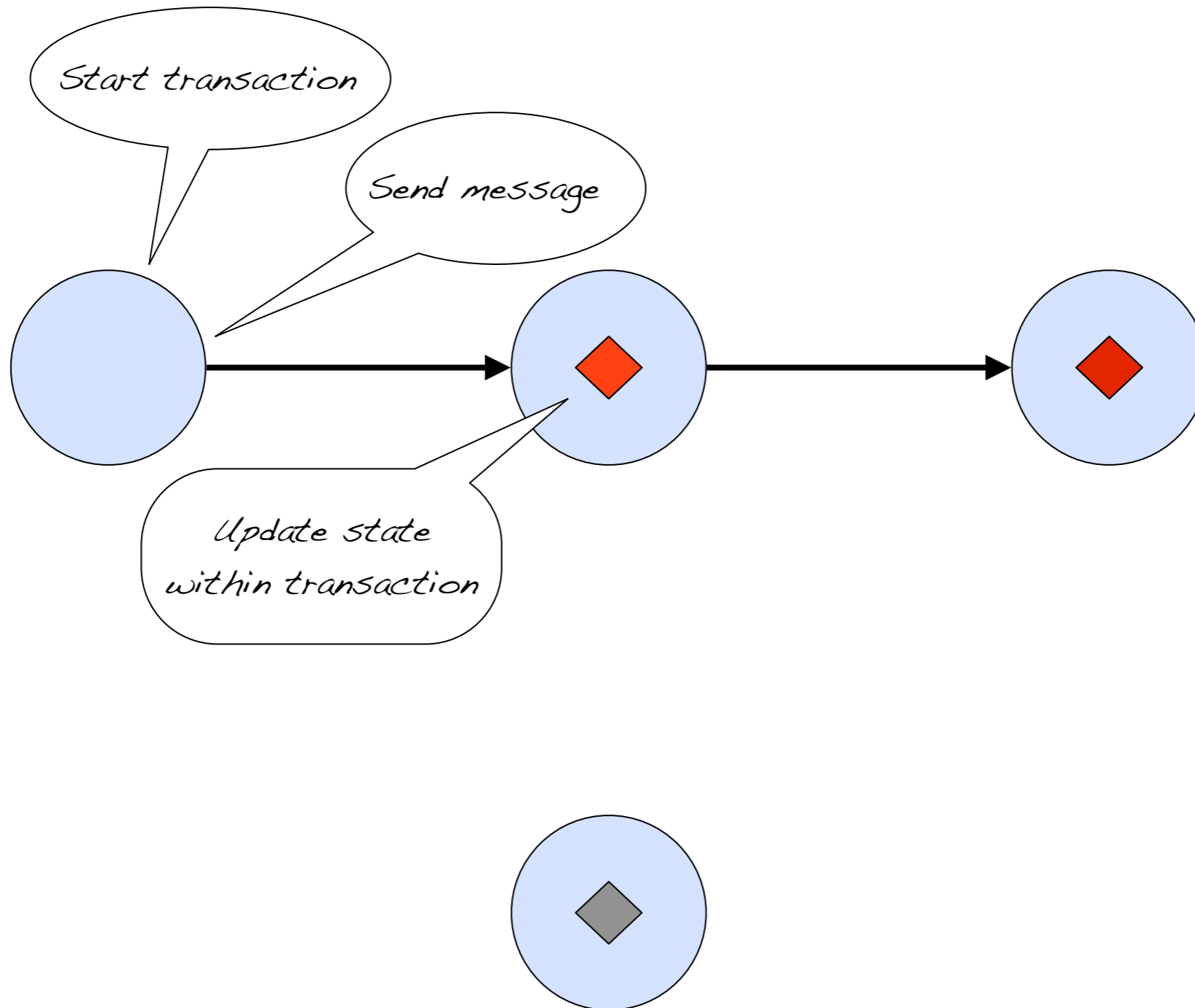
Transactors



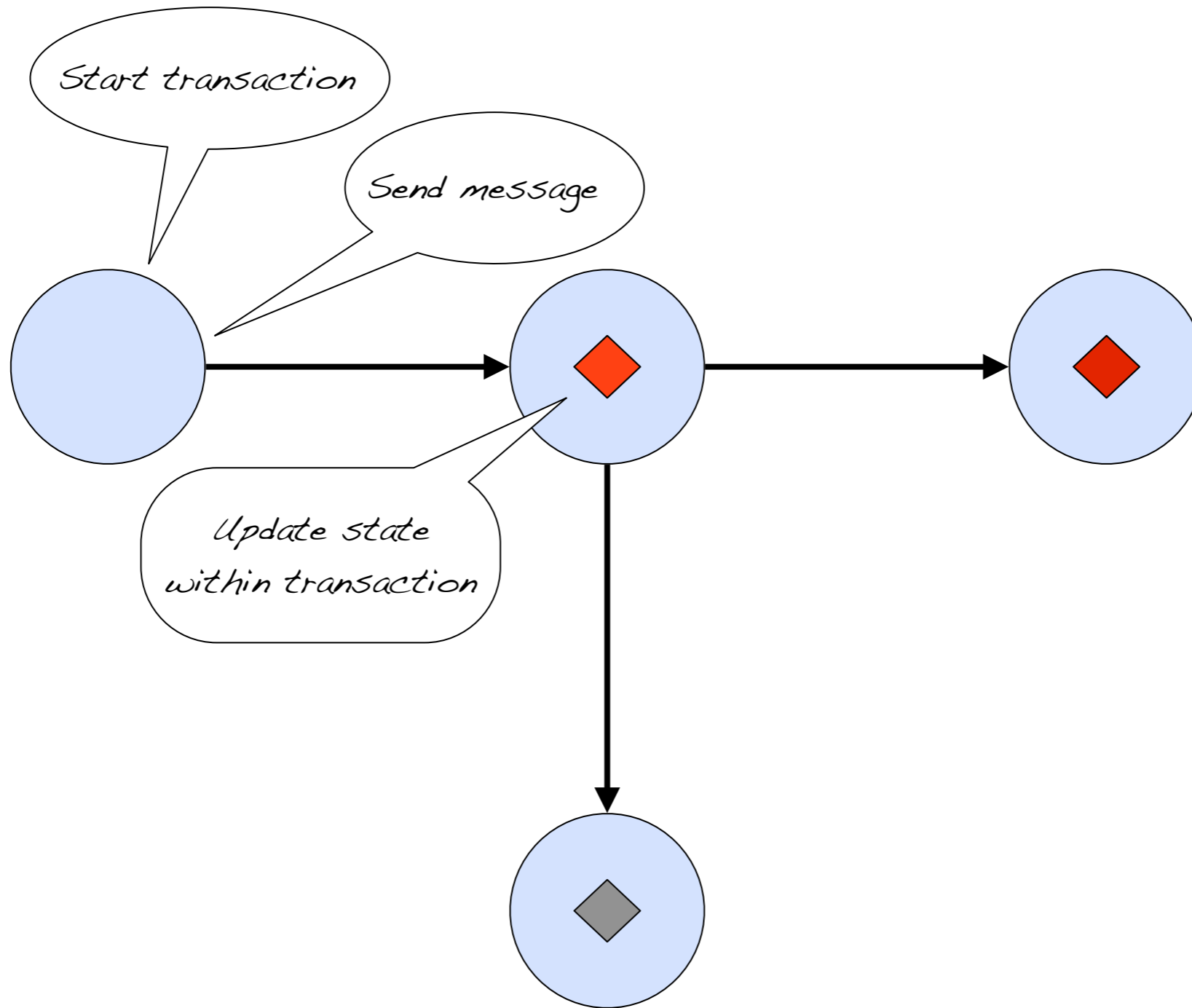
Transactors



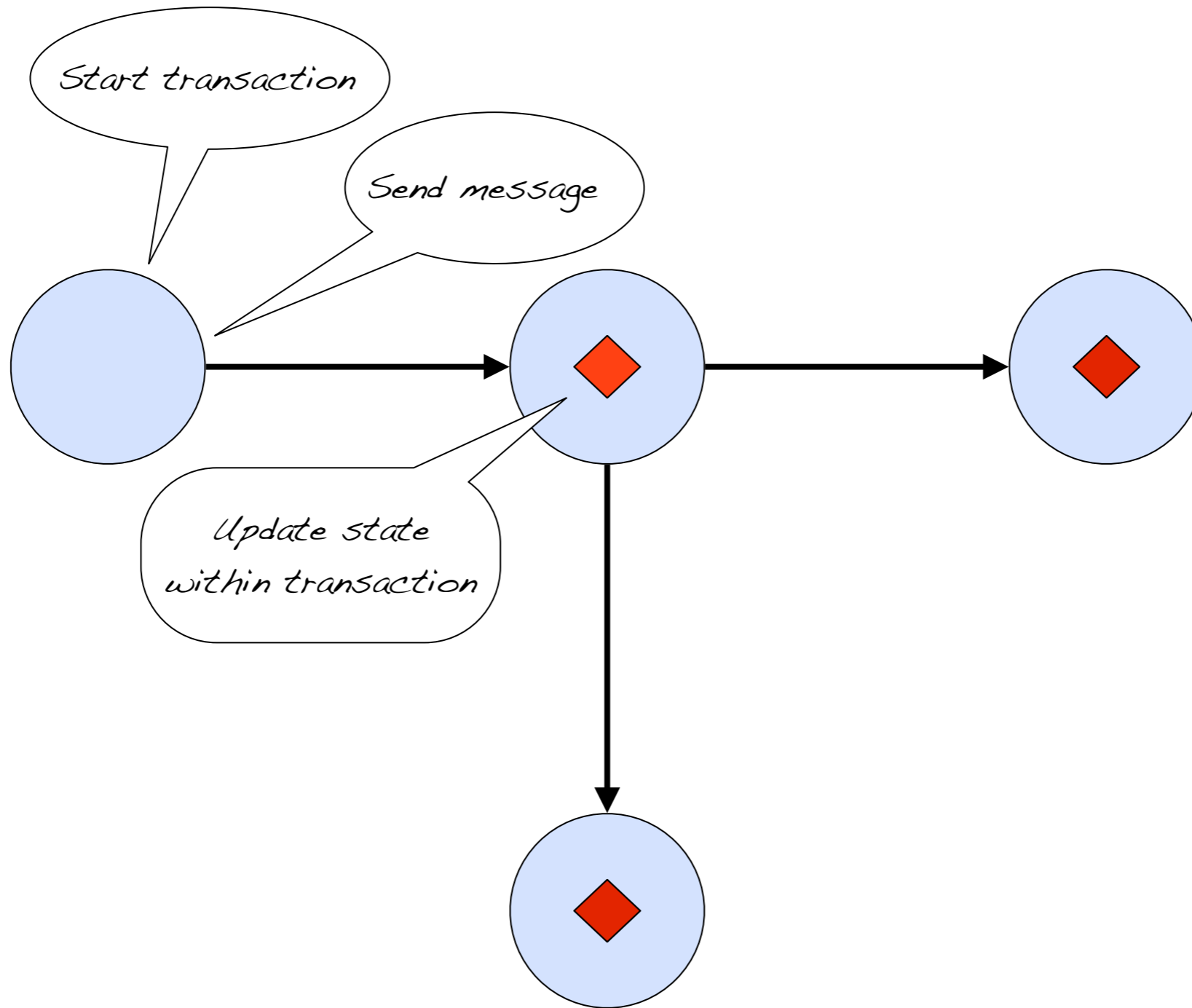
Transactors



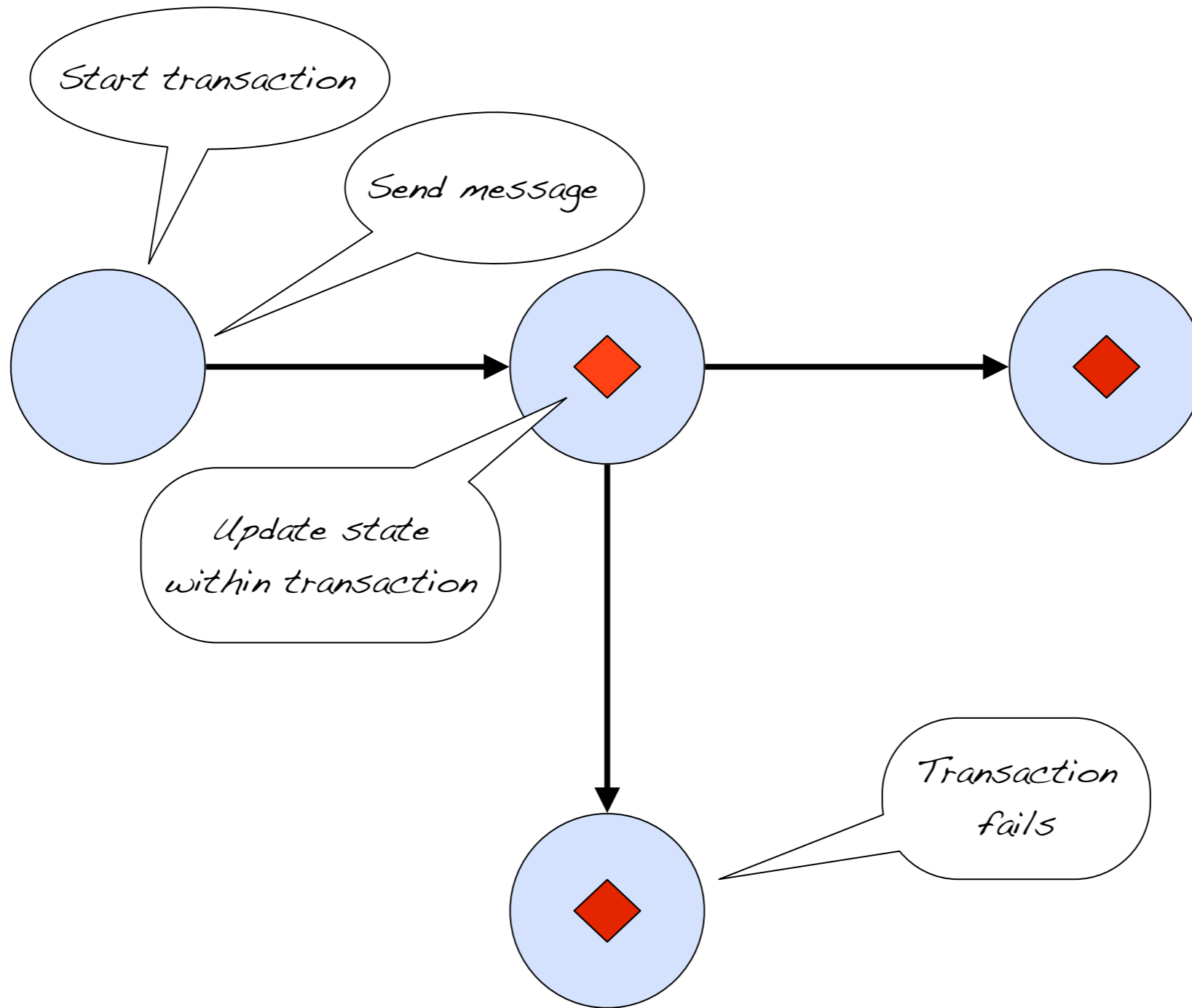
Transactors



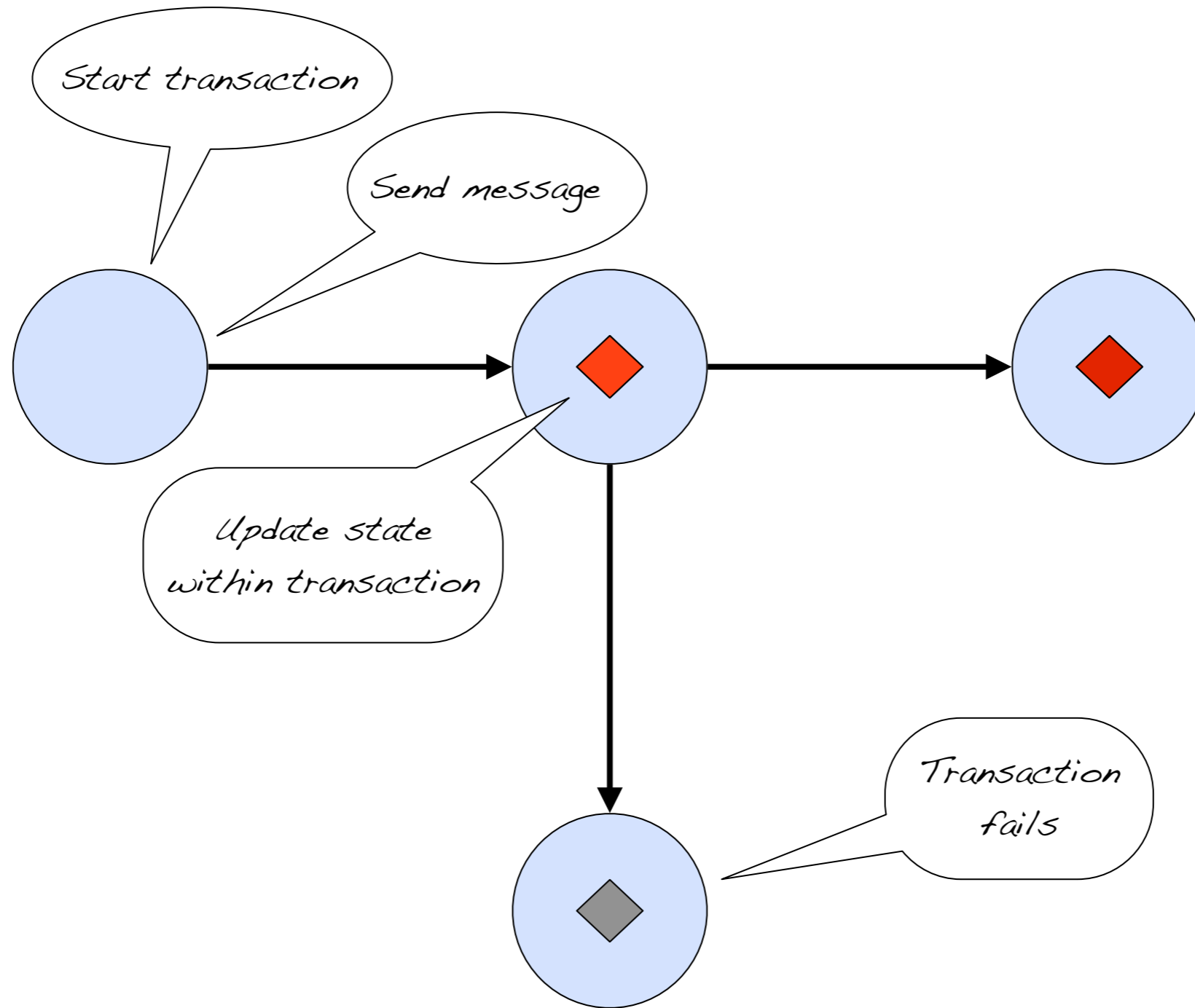
Transactors



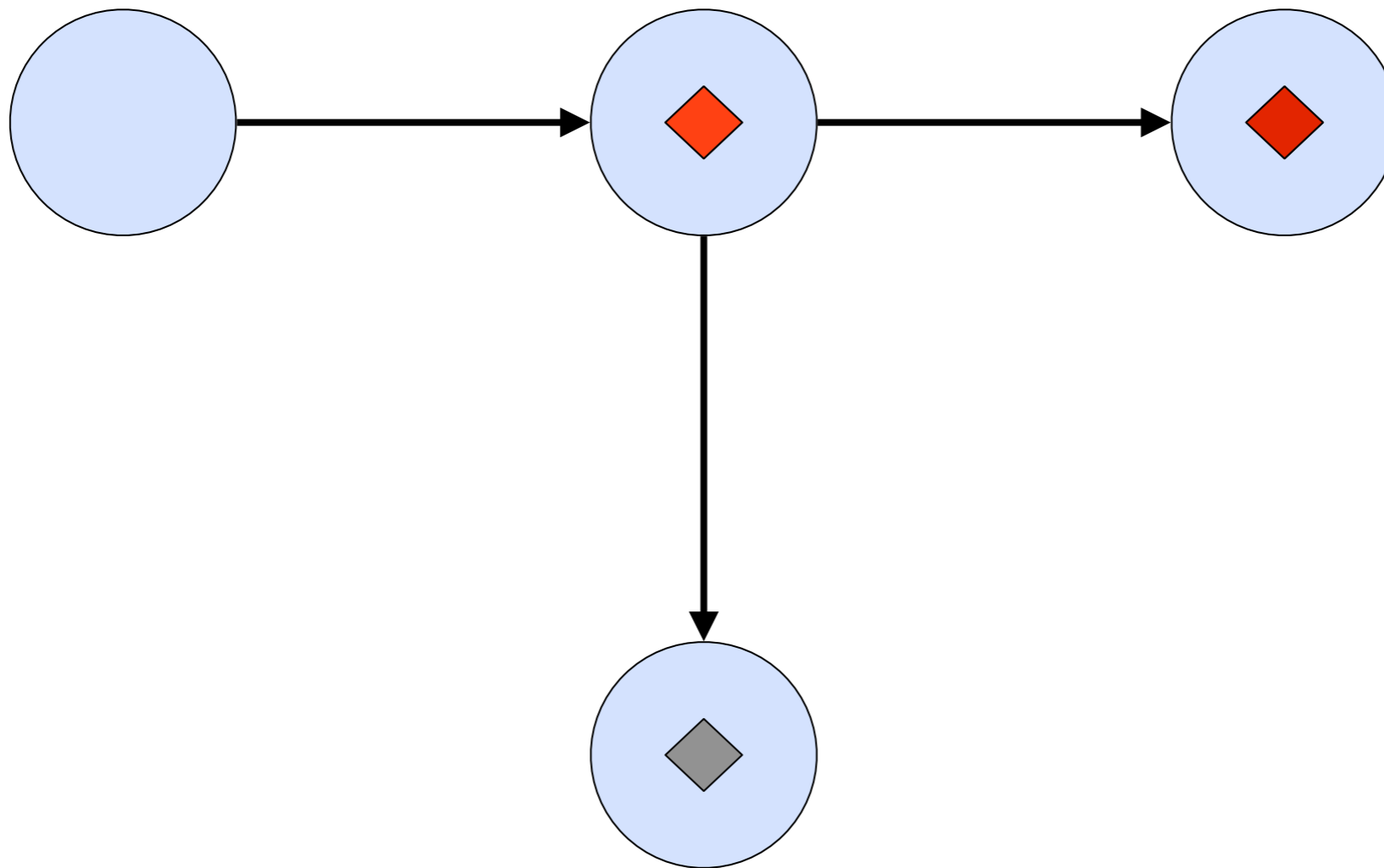
Transactors



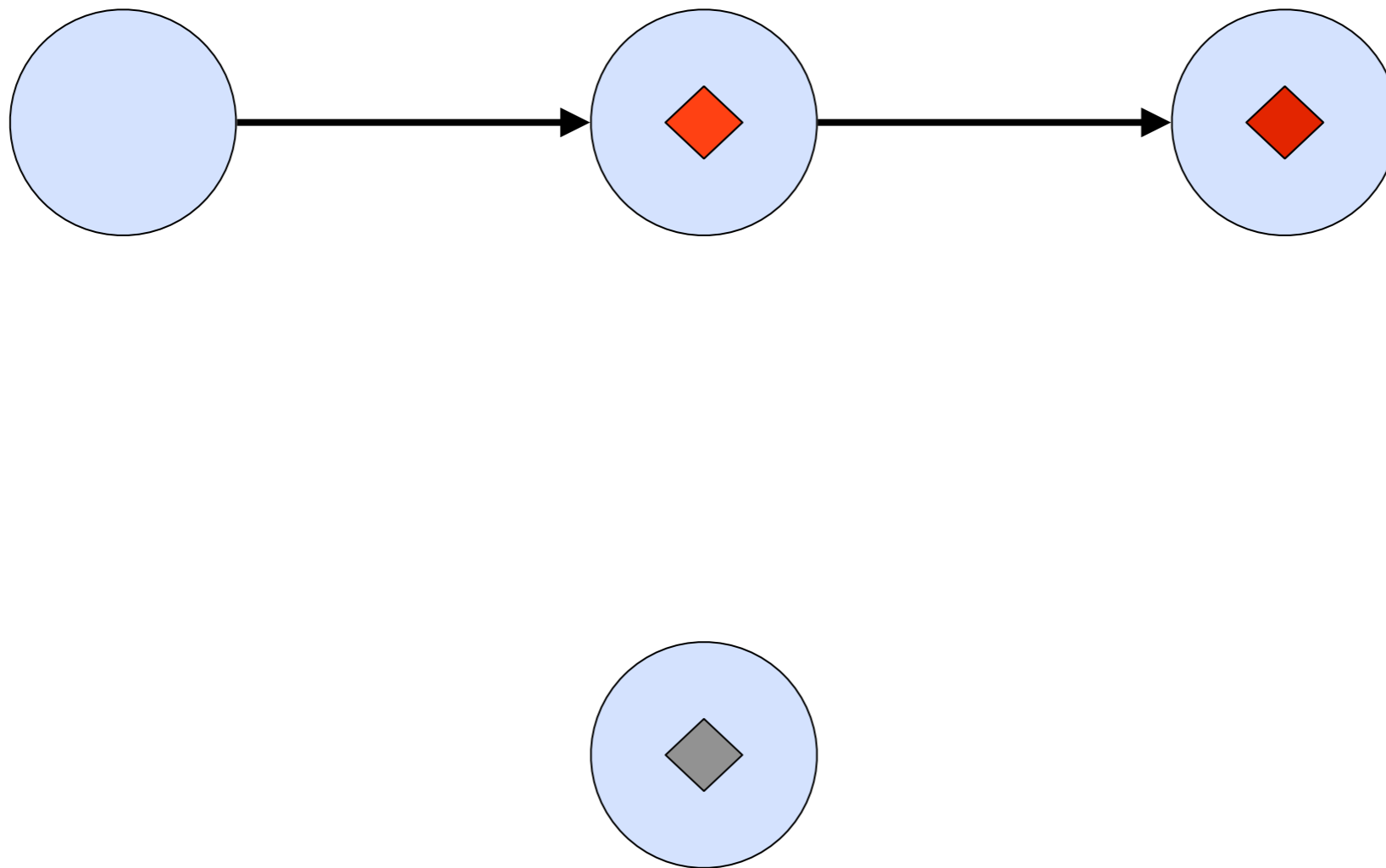
Transactors



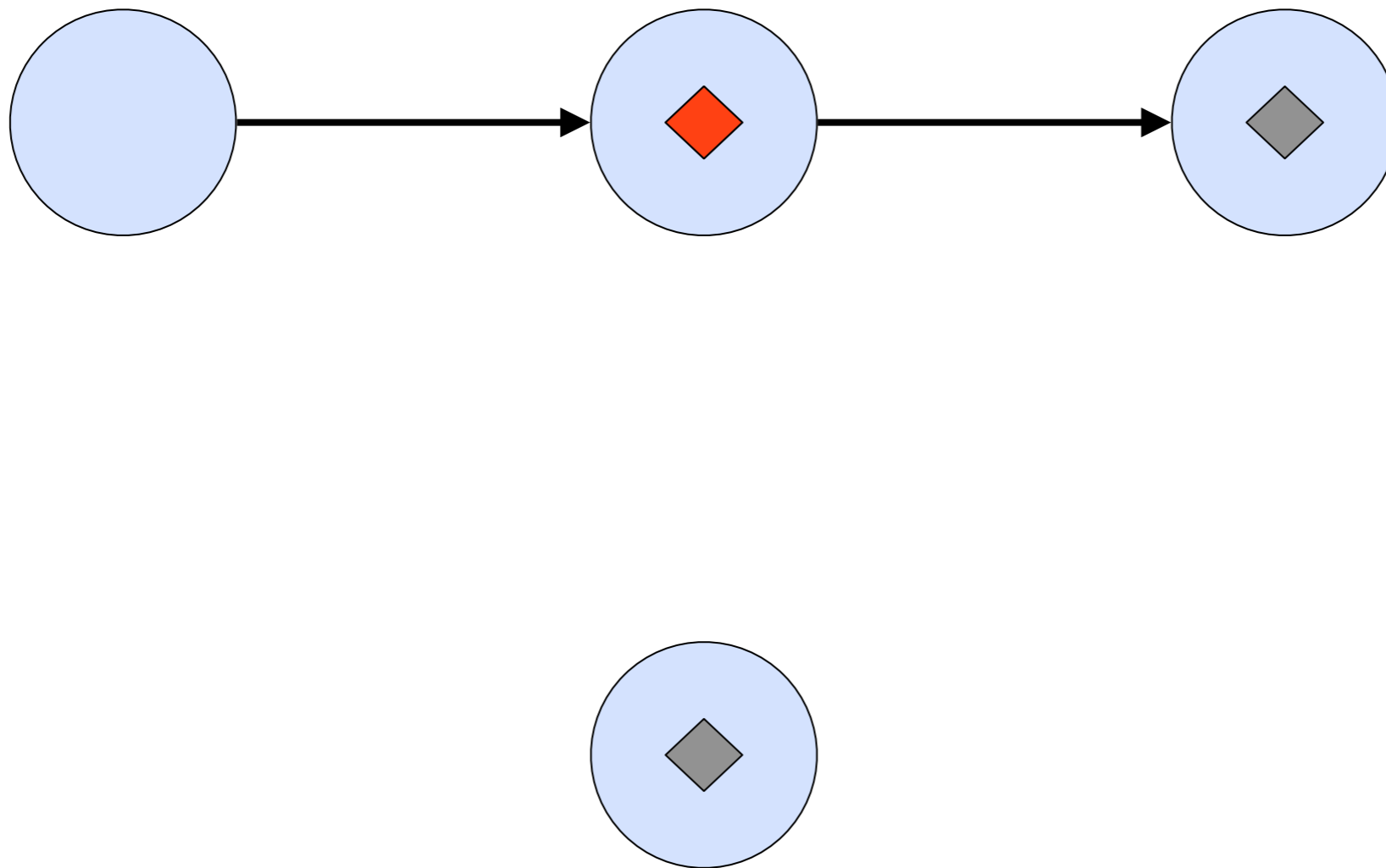
Transactors



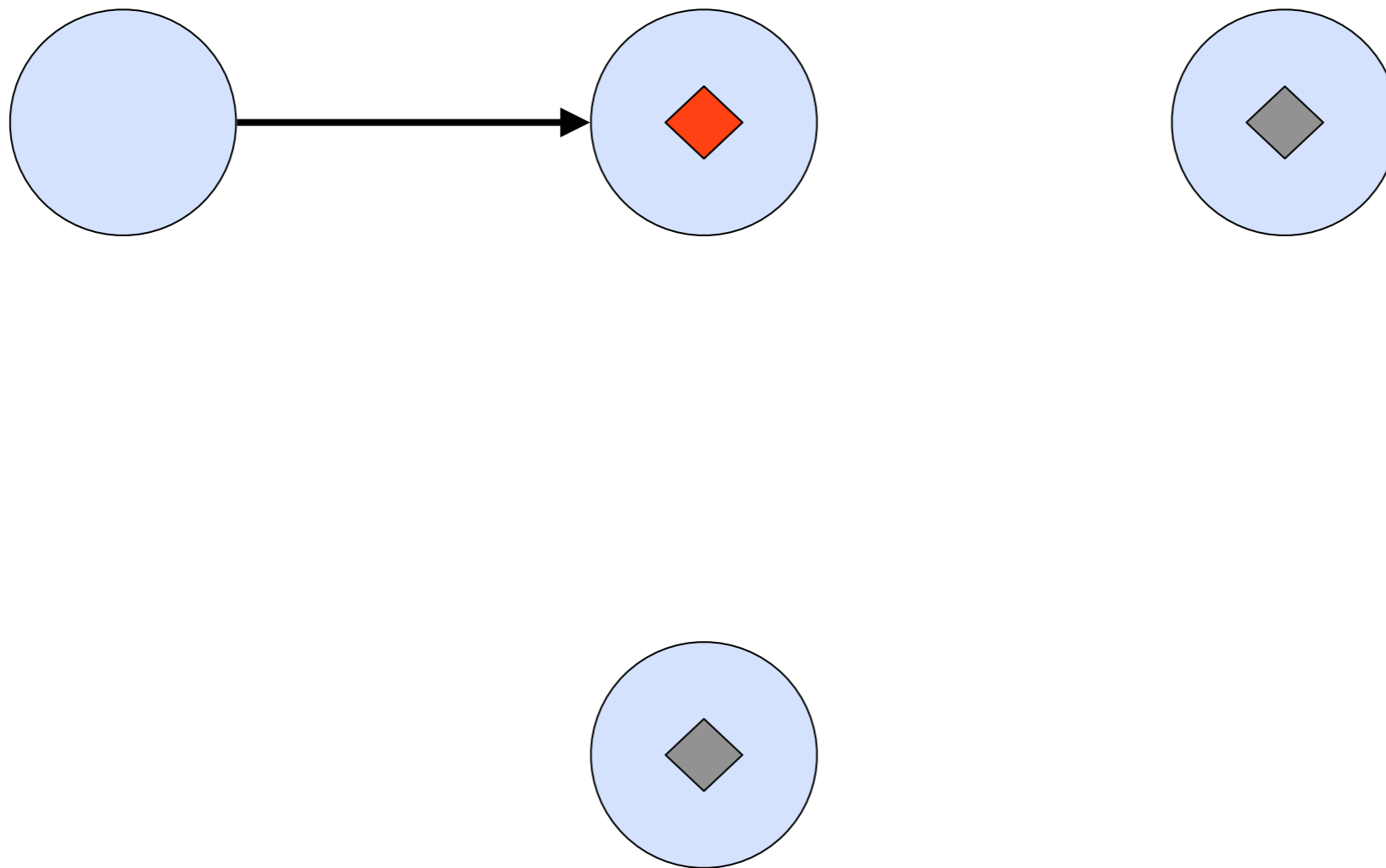
Transactors



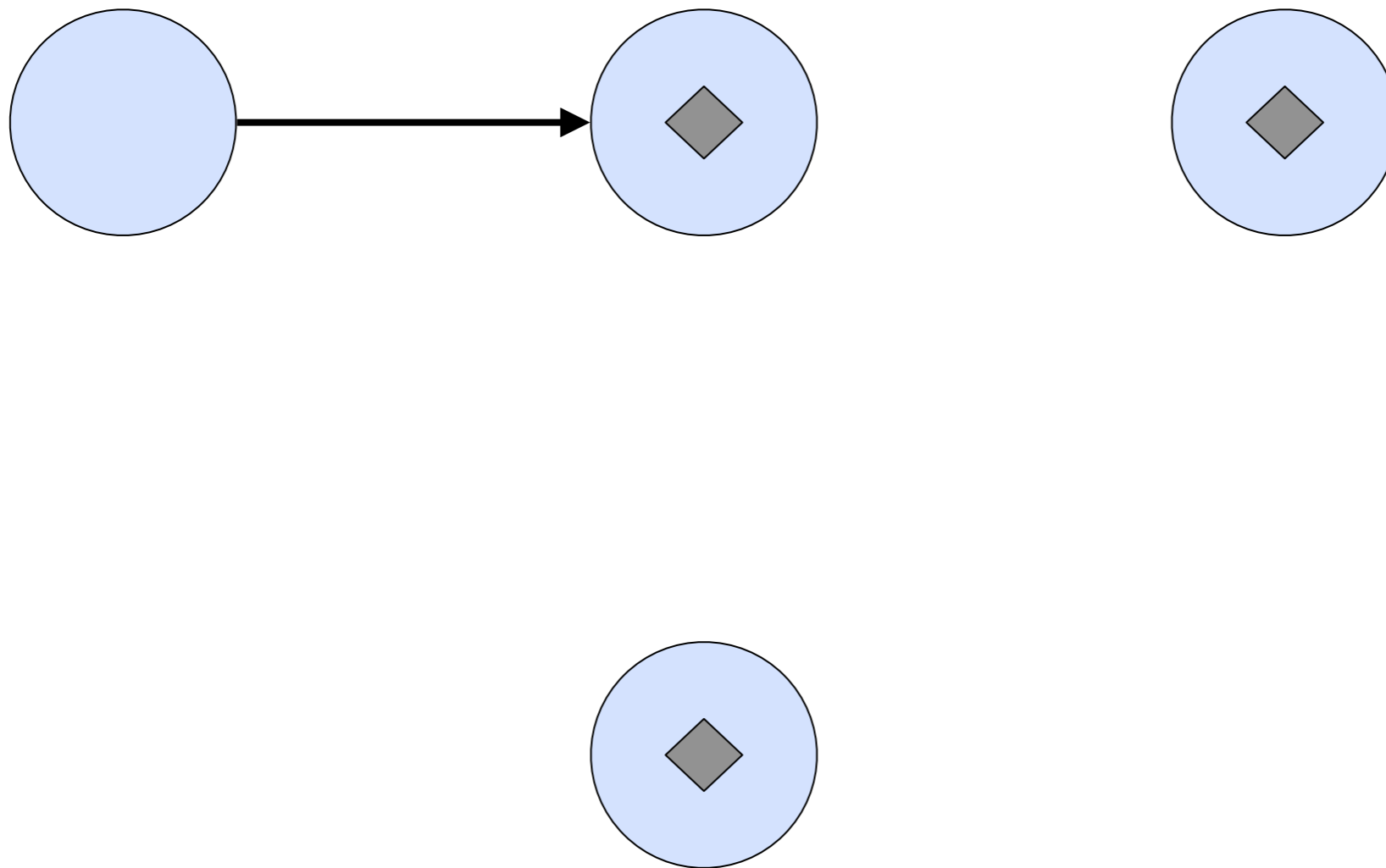
Transactors



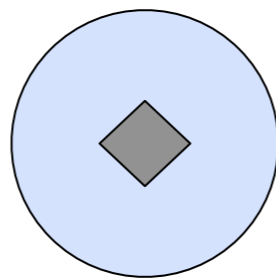
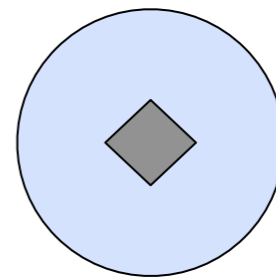
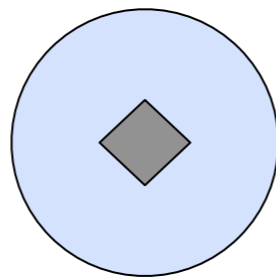
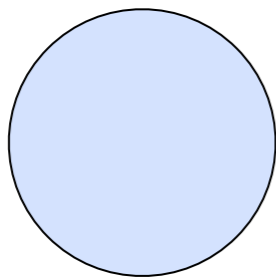
Transactors



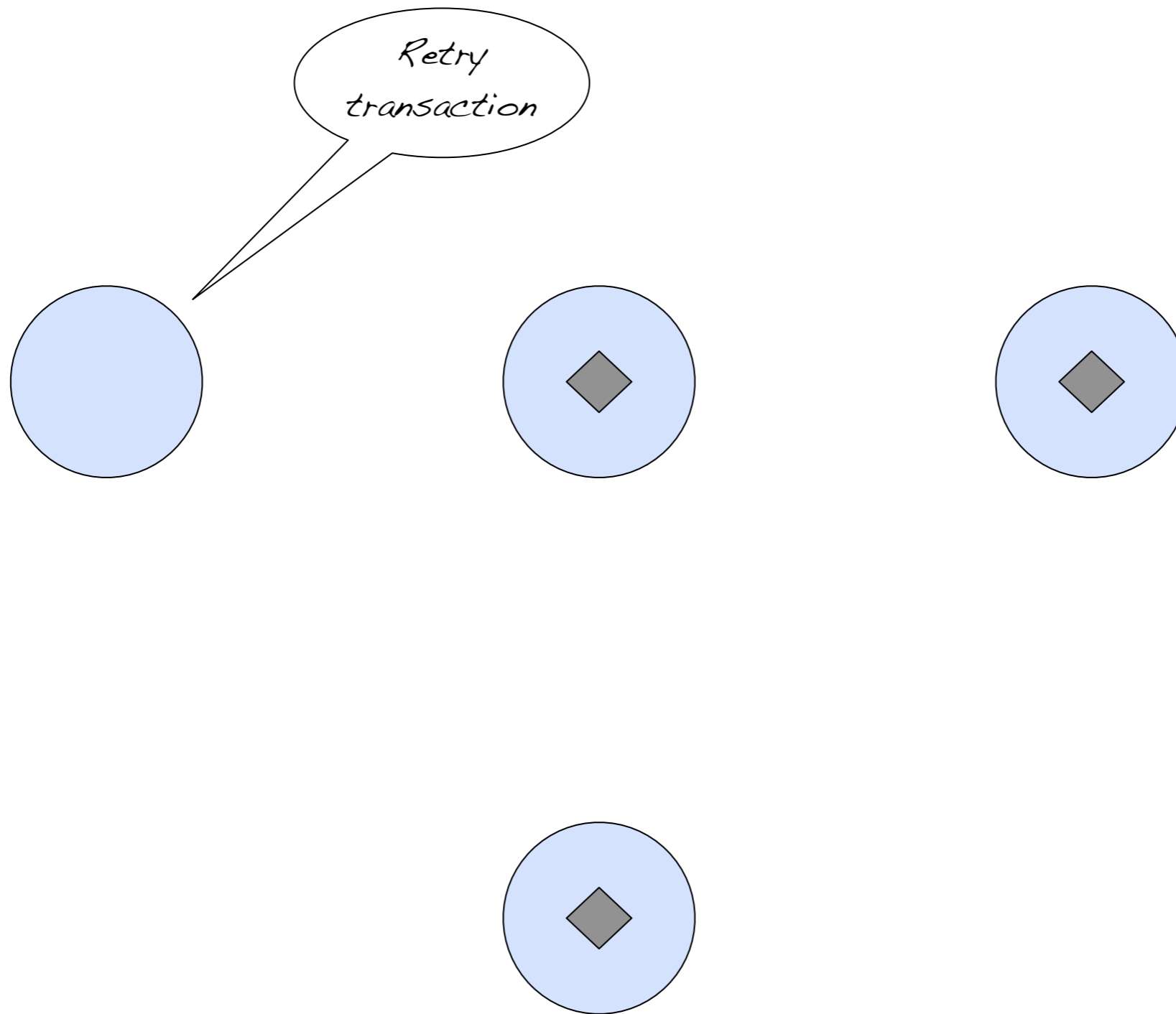
Transactors



Transactors



Transactors



Modules

Akka Spring

Spring integration

```
<beans>  
  <akka:active-object id="myActiveObject"  
    target="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:active-objects>
    <akka:active-object target="com.biz.MyPOJO"
      lifecycle="permanent" timeout="1000">
      <akka:restart-callbacks pre="preRestart" post="postRestart"/>
    </akka:active-object>
  </akka:active-objects>
</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 = new CometProducer
```

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

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

Deployment

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

...and much more

Persistence

REST

Security

Comet

Lift

Guice

Learn more

<http://akkasource.org>

Professional help

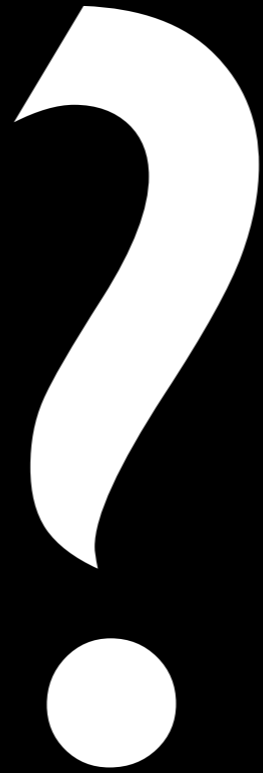
Support packages

Consulting Training Mentoring

<http://scalablesolutions.se>

jonas@jonasboner.com

EOOF



Managed References

```
val usersRef =  
    TransactionalRef(Map[String, User]())  
  
for (users <- usersRef) {  
    users + (name -> user)  
}  
  
val user = for (users <- usersRef) yield {  
    users(name)  
}
```

Managed References

```
for {  
  users <- usersRef  
  user <- users  
  roles <- rolesRef  
  role <- roles  
  if user.hasRole(role)  
} {  
  ... // do stuff  
}
```

STM:API

```
import se.scalablesolutions.akka.stm.Transaction._  
  
atomically {  
  .. // try to do something  
} orElse {  
  .. // if tx clash; try do do something else  
}
```

Putting it together

```
for {  
  tx      <- Transaction()  
  users   <- usersRef  
  user    <- users  
  roles   <- rolesRef  
  role    <- roles  
  if user.hasRole(role)  
} {  
  ... // do stuff  
}
```