Akka:

Simpler Concurrency, Scalability & Fault-tolerance through Actors

Jonas Bonér

Viktor Klang

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

Introducing

STM Actors Agents

Dataflow Distributed Open Source

RESTful Secure Persistent

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

Microbenchmark

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

Akka 0.8.1	**3815**
Scala Actors 2.8.0.Beta1 (react)	**16086**

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

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()
}
```

Send: !

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

Send: !!

```
// uses Future with default timeout
val result: Option[..] = actor !! Message
```

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)
  }
}
```

Akka Dispatchers

Dispatchers

```
class Dispatchers {
  def newThreadBasedDispatcher(actor: Actor)

  def newExecutorBasedEventDrivenDispatcher

  def newExecutorBasedEventDrivenWorkStealingDispatcher

  ... // etc
}
```

Set dispatcher

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

  ...
}

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

Let it crash

fault-tolerance

Stolen from

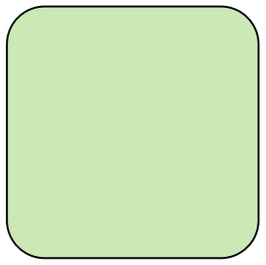
Erlang

9

nines

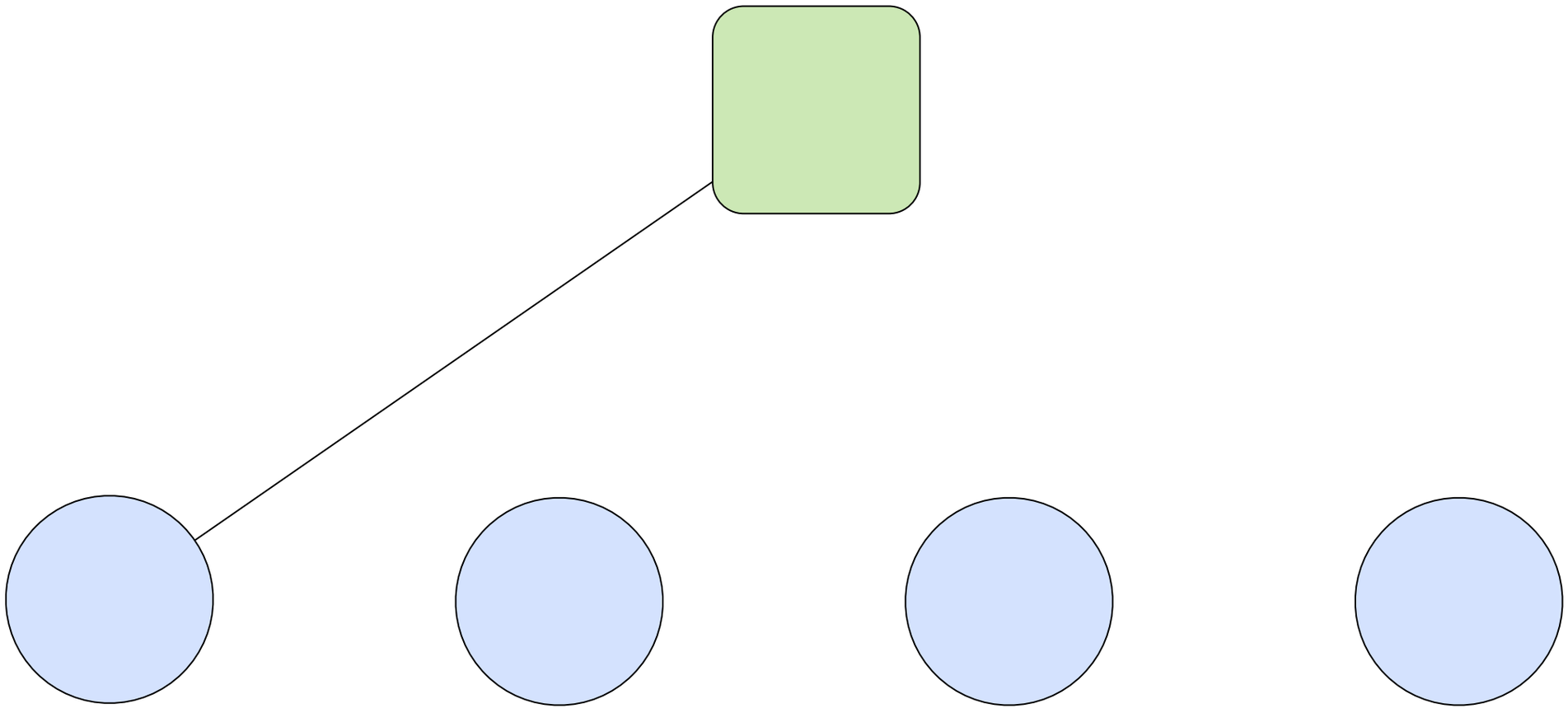
OneForOne

fault handling strategy



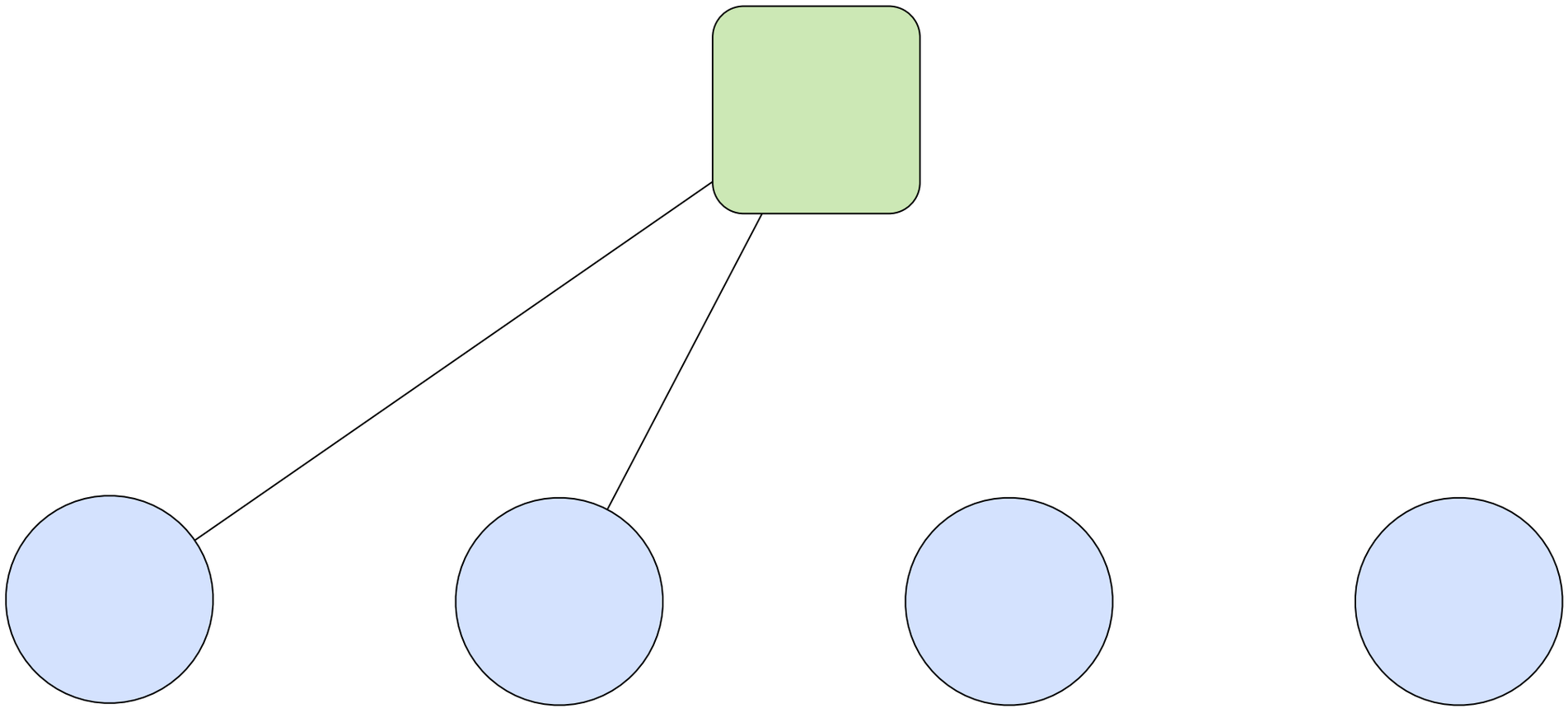
OneForOne

fault handling strategy



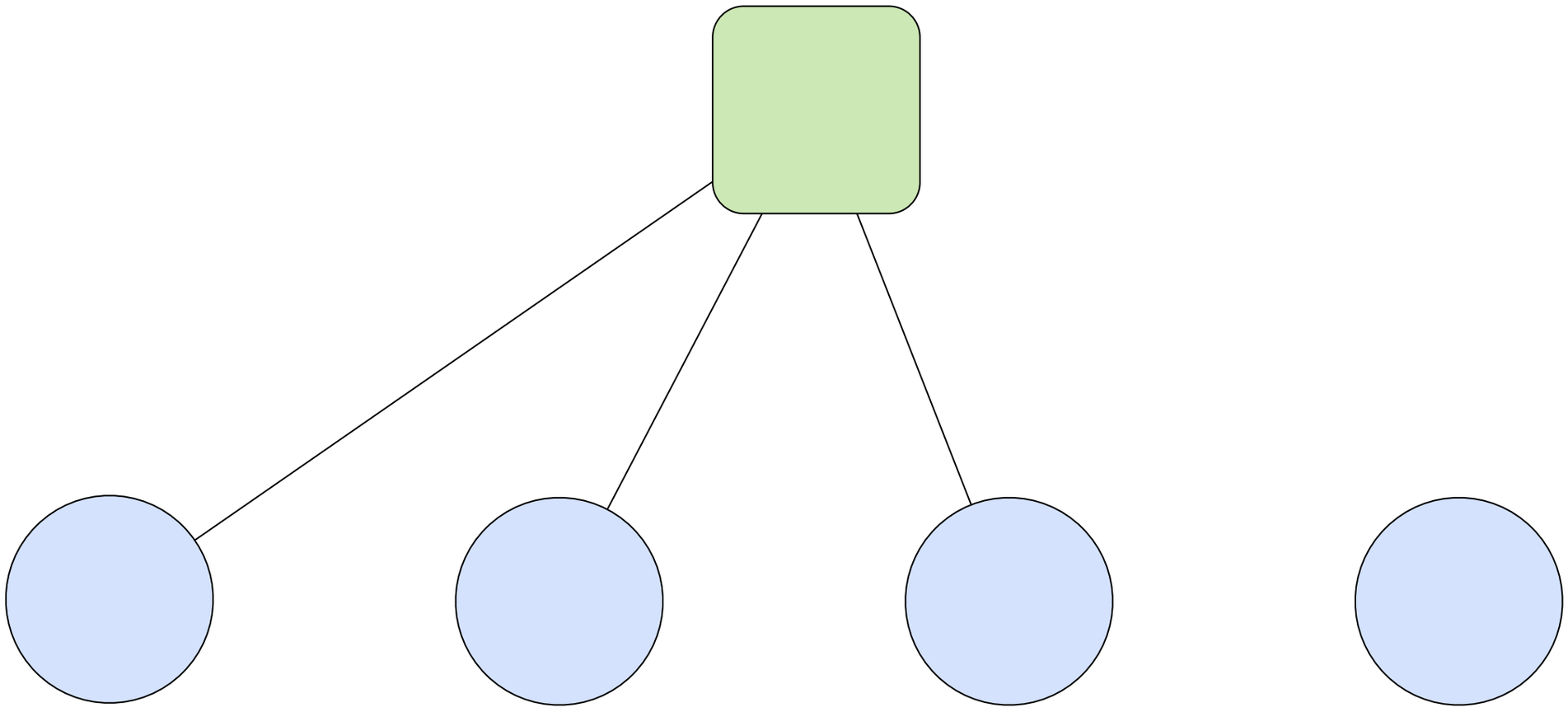
OneForOne

fault handling strategy



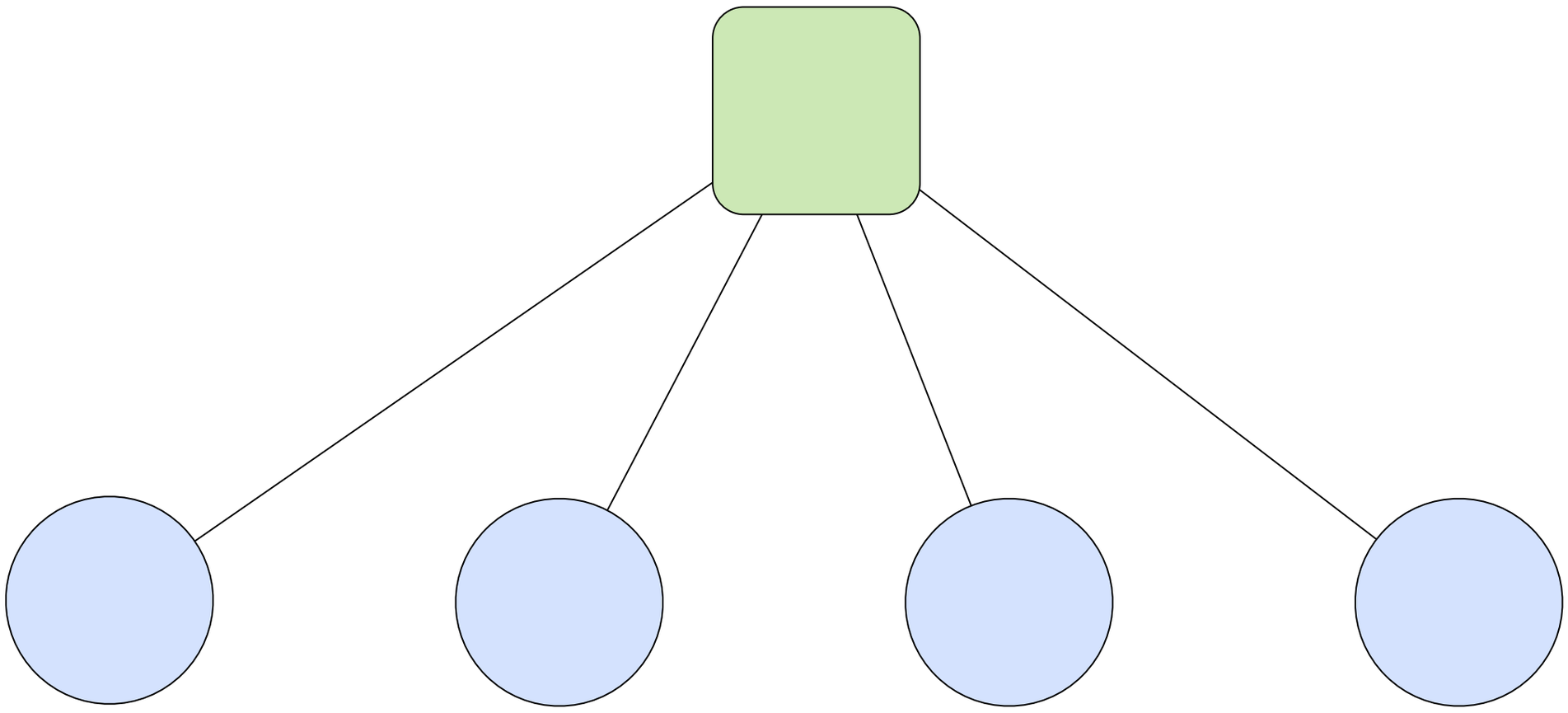
OneForOne

fault handling strategy



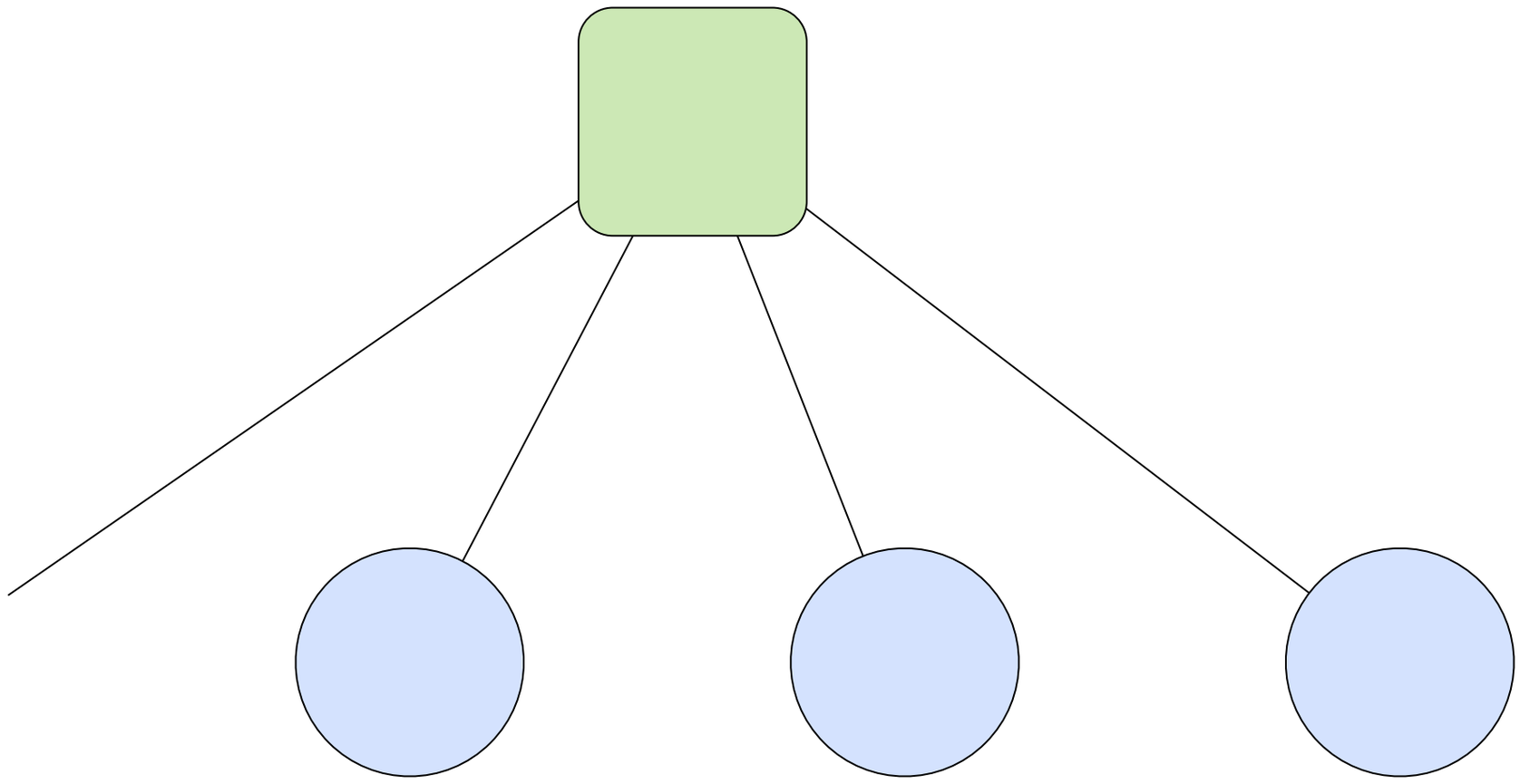
OneForOne

fault handling strategy



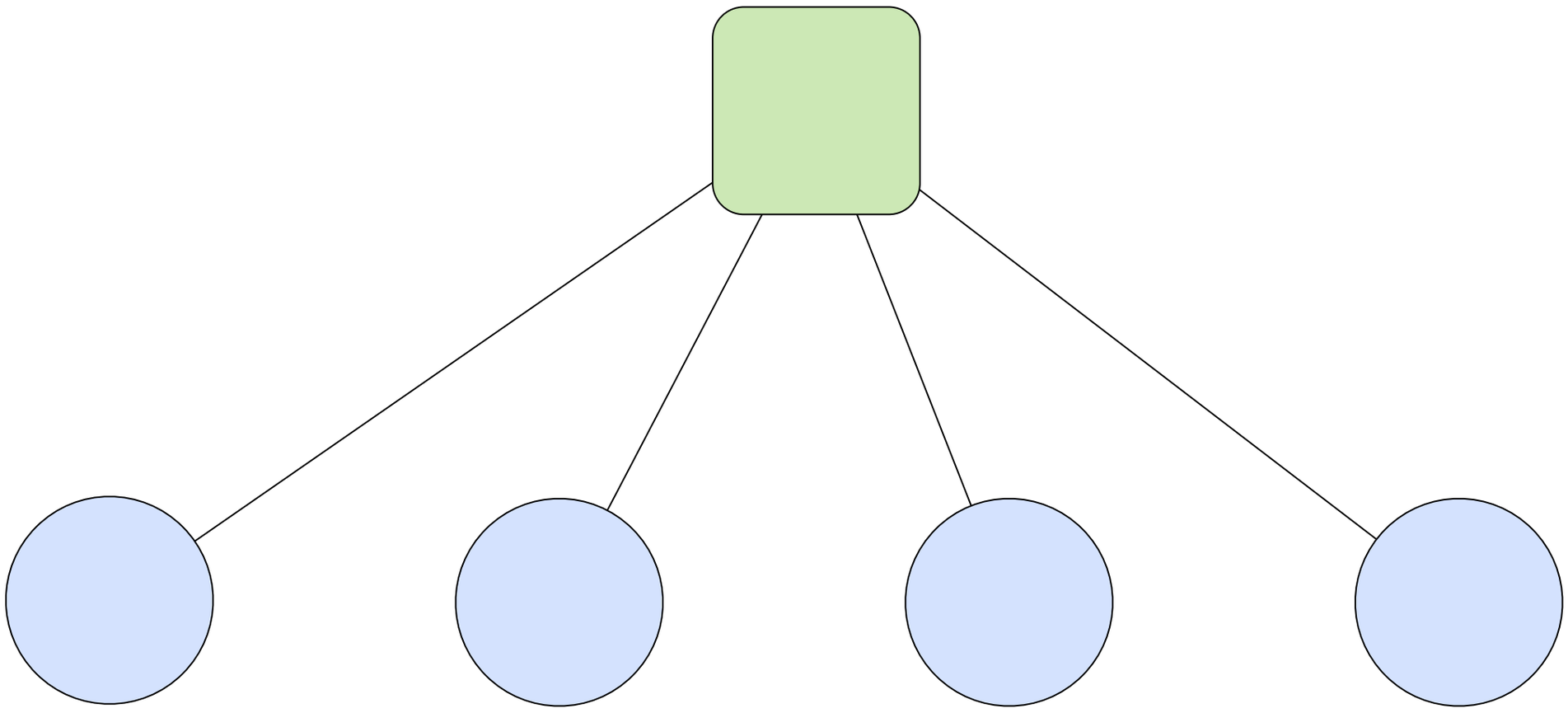
OneForOne

fault handling strategy



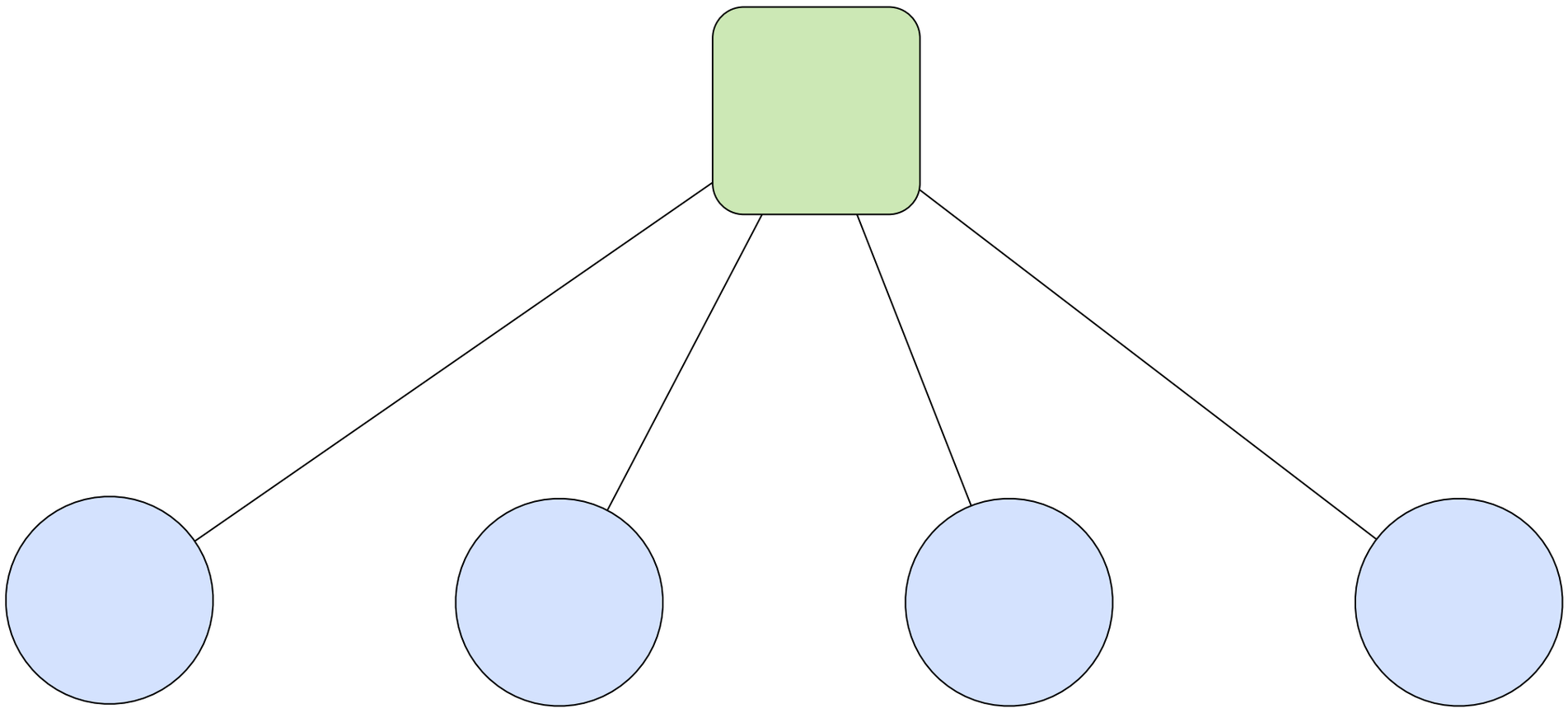
OneForOne

fault handling strategy



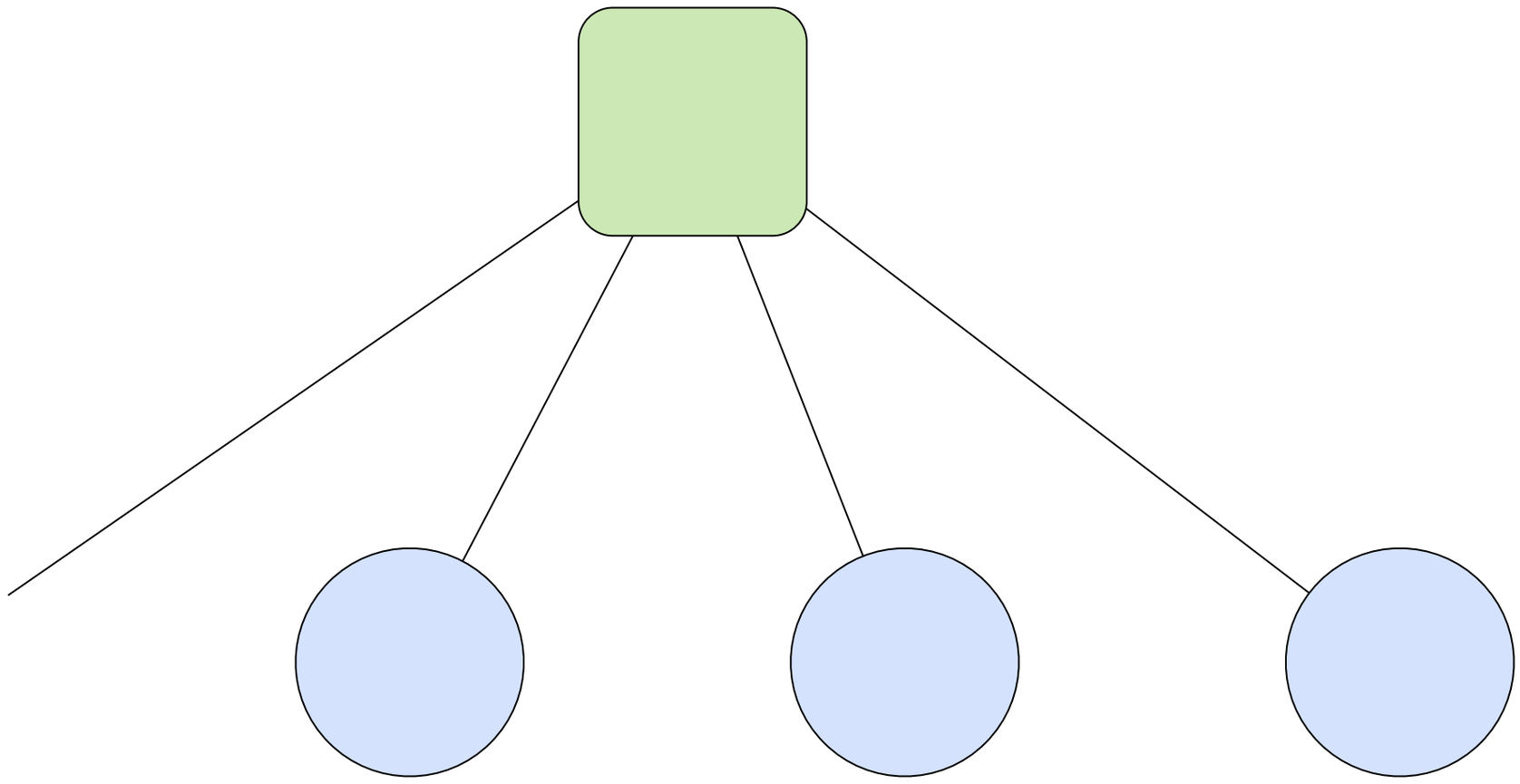
AllForOne

fault handling strategy



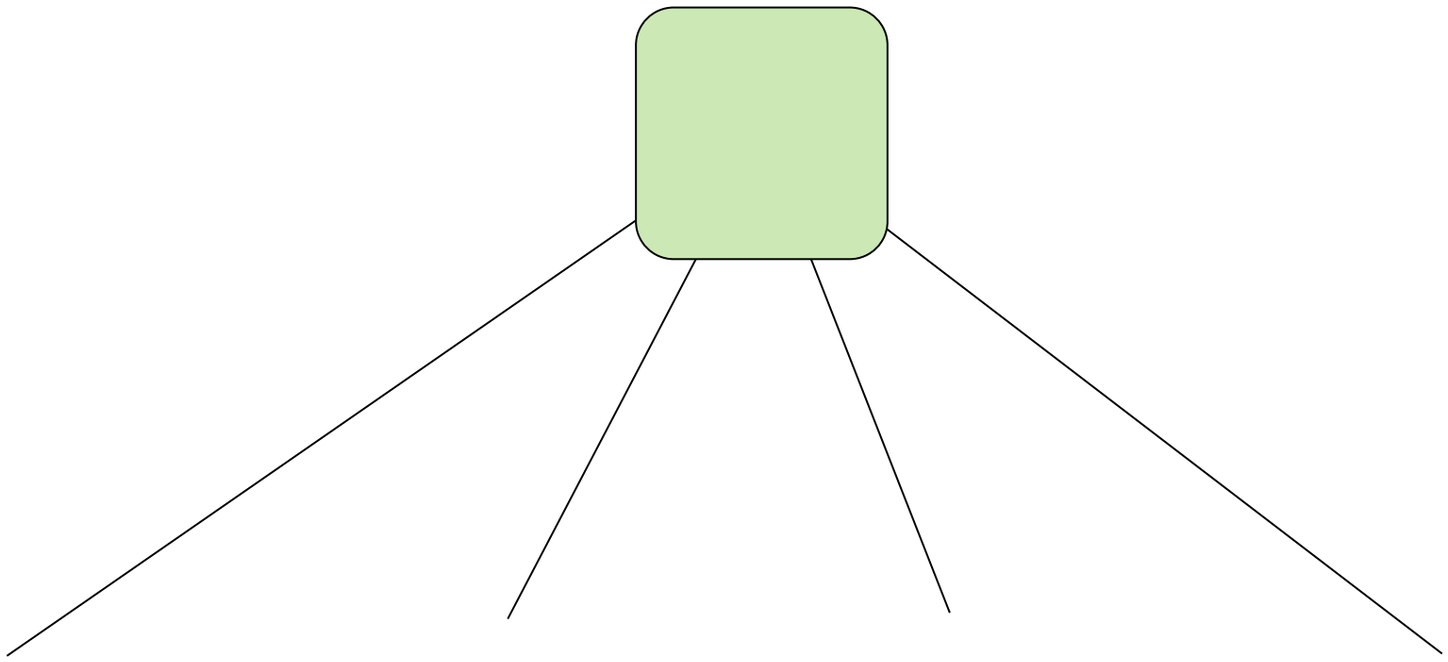
AllForOne

fault handling strategy



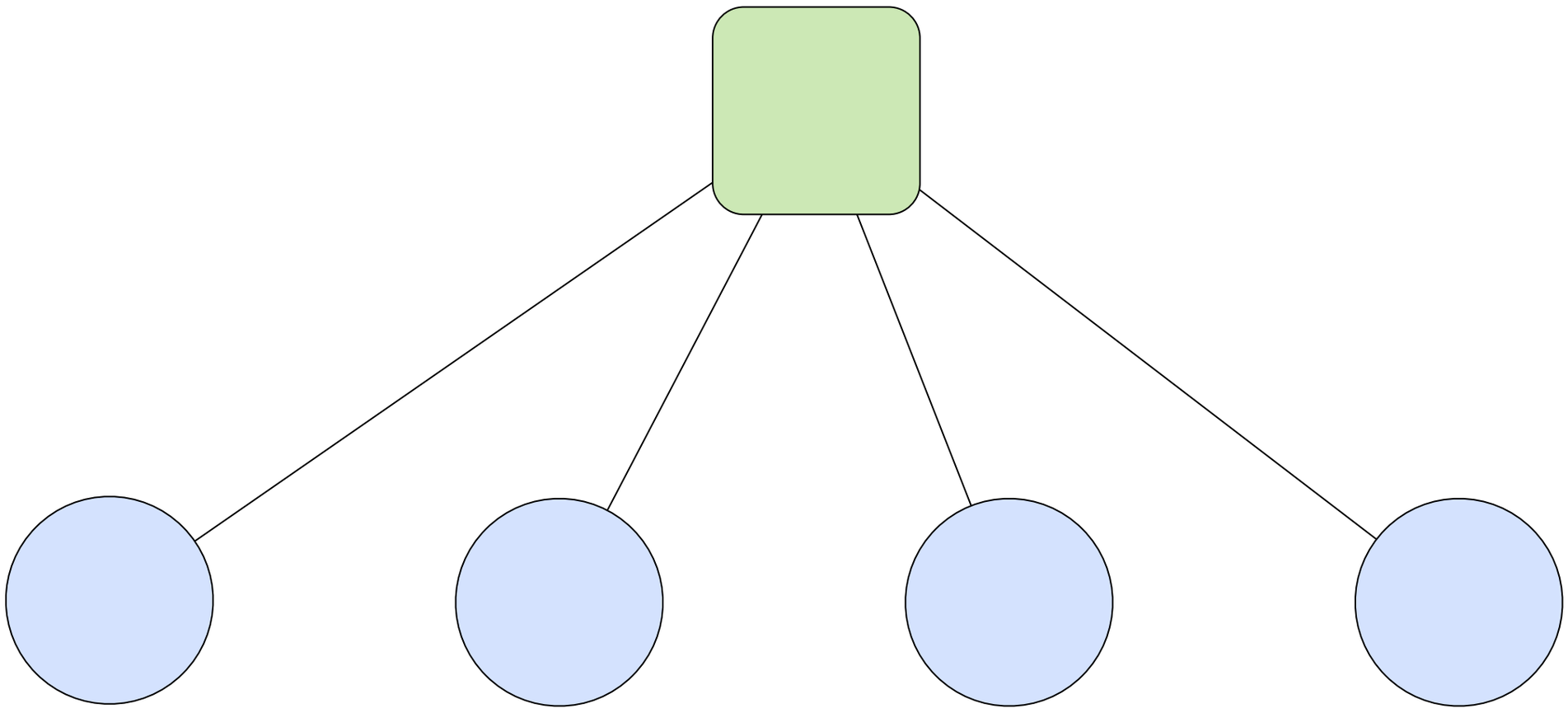
AllForOne

fault handling strategy

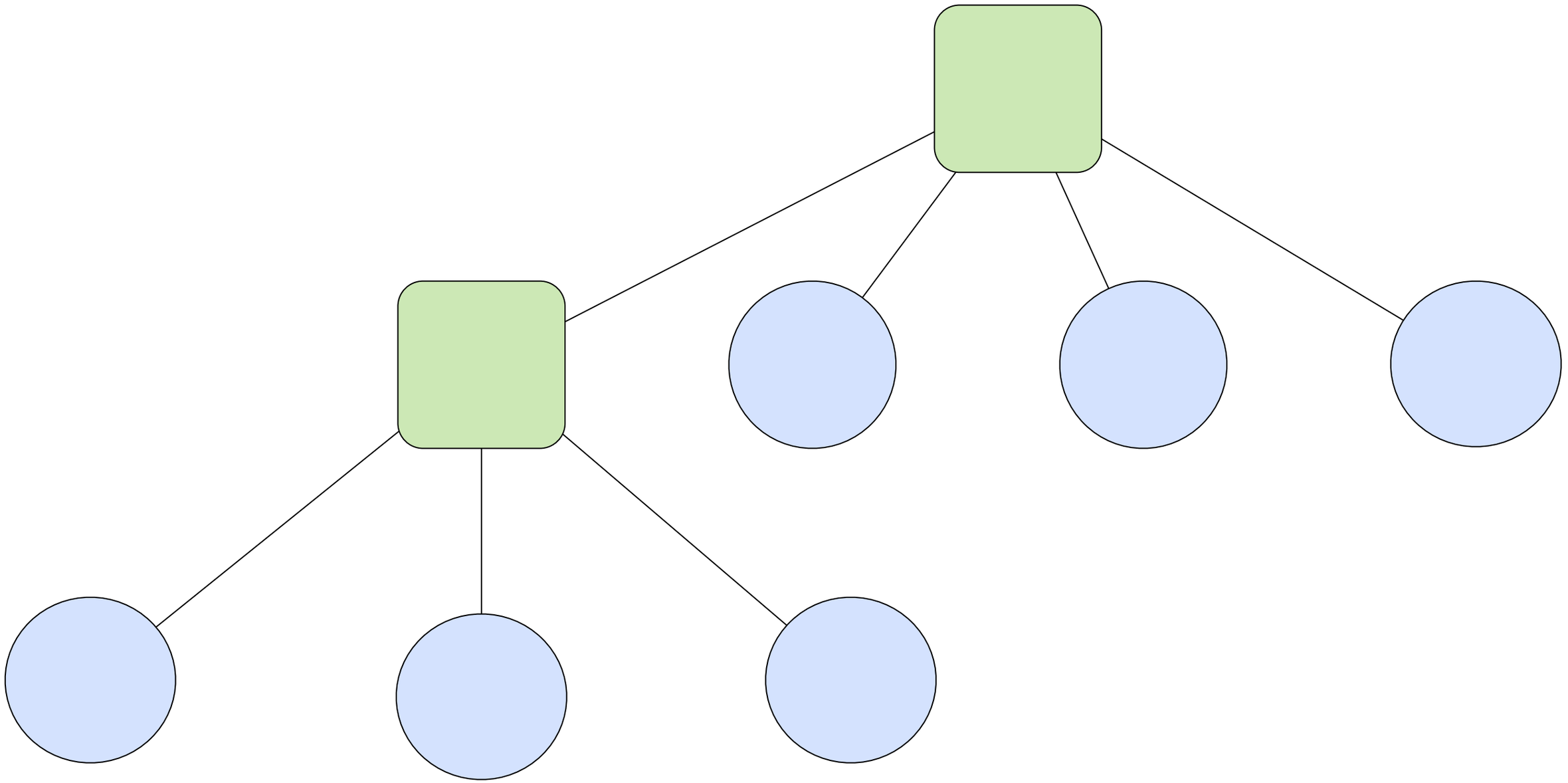


AllForOne

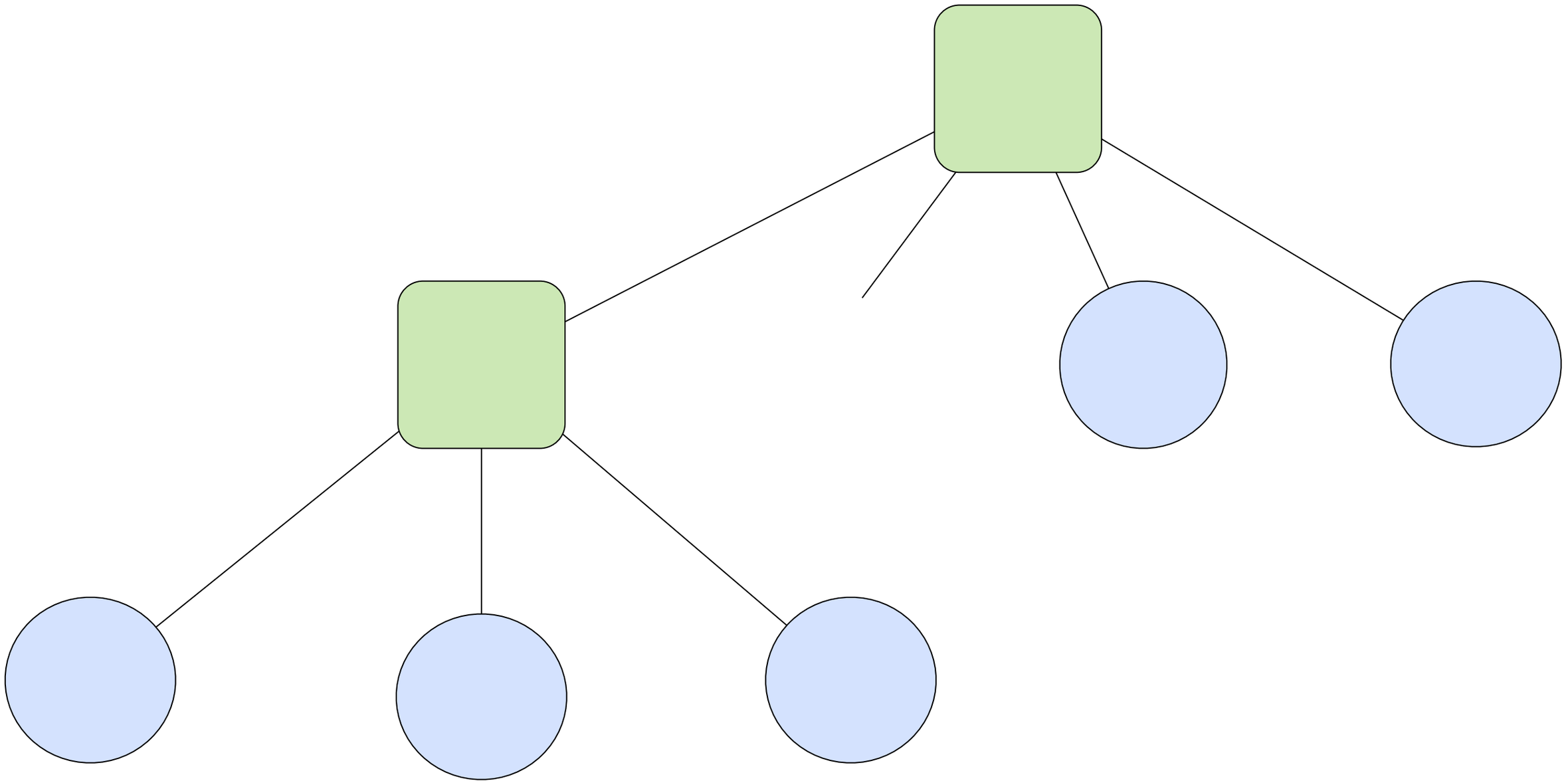
fault handling strategy



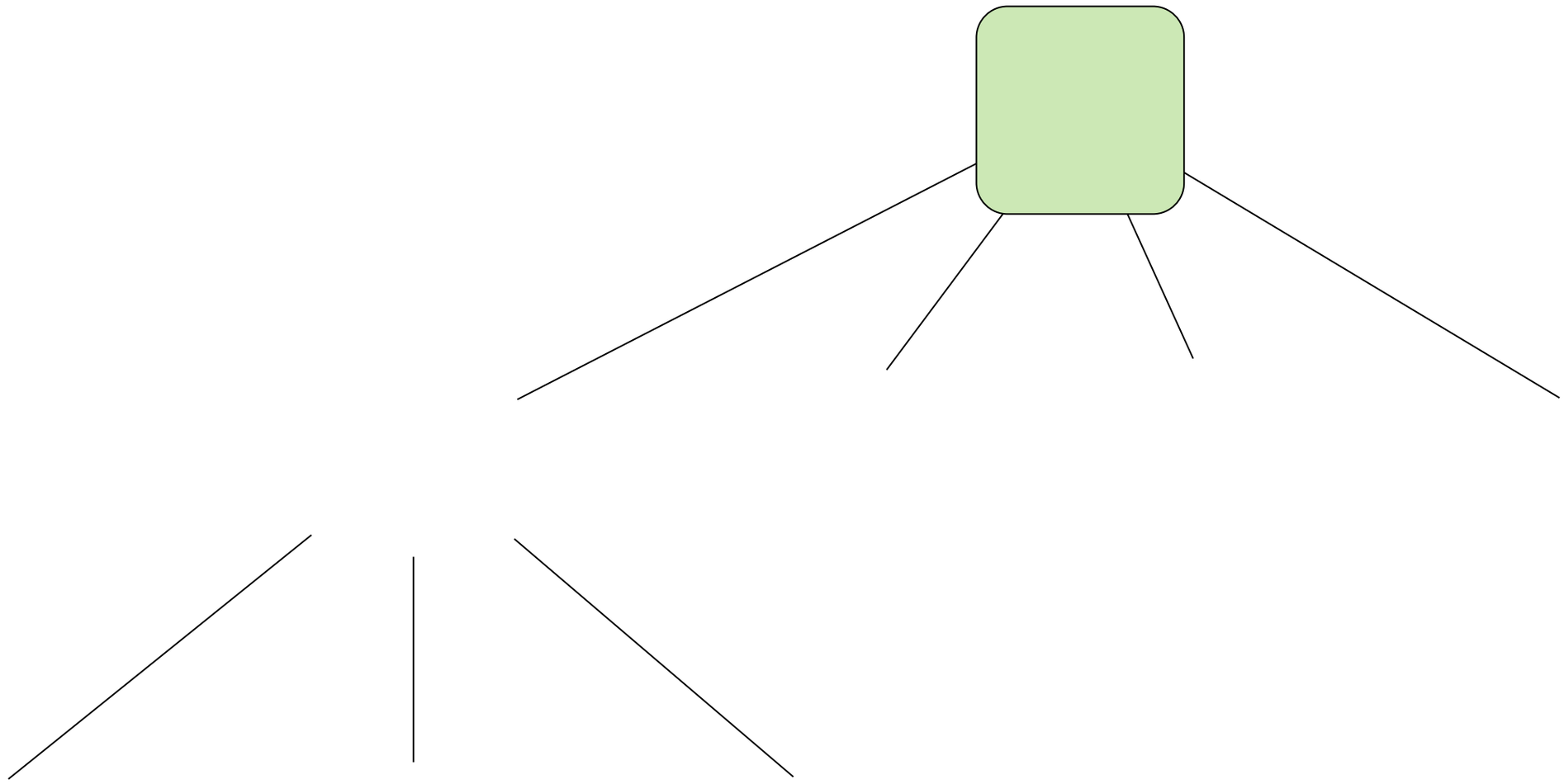
Supervisor hierarchies



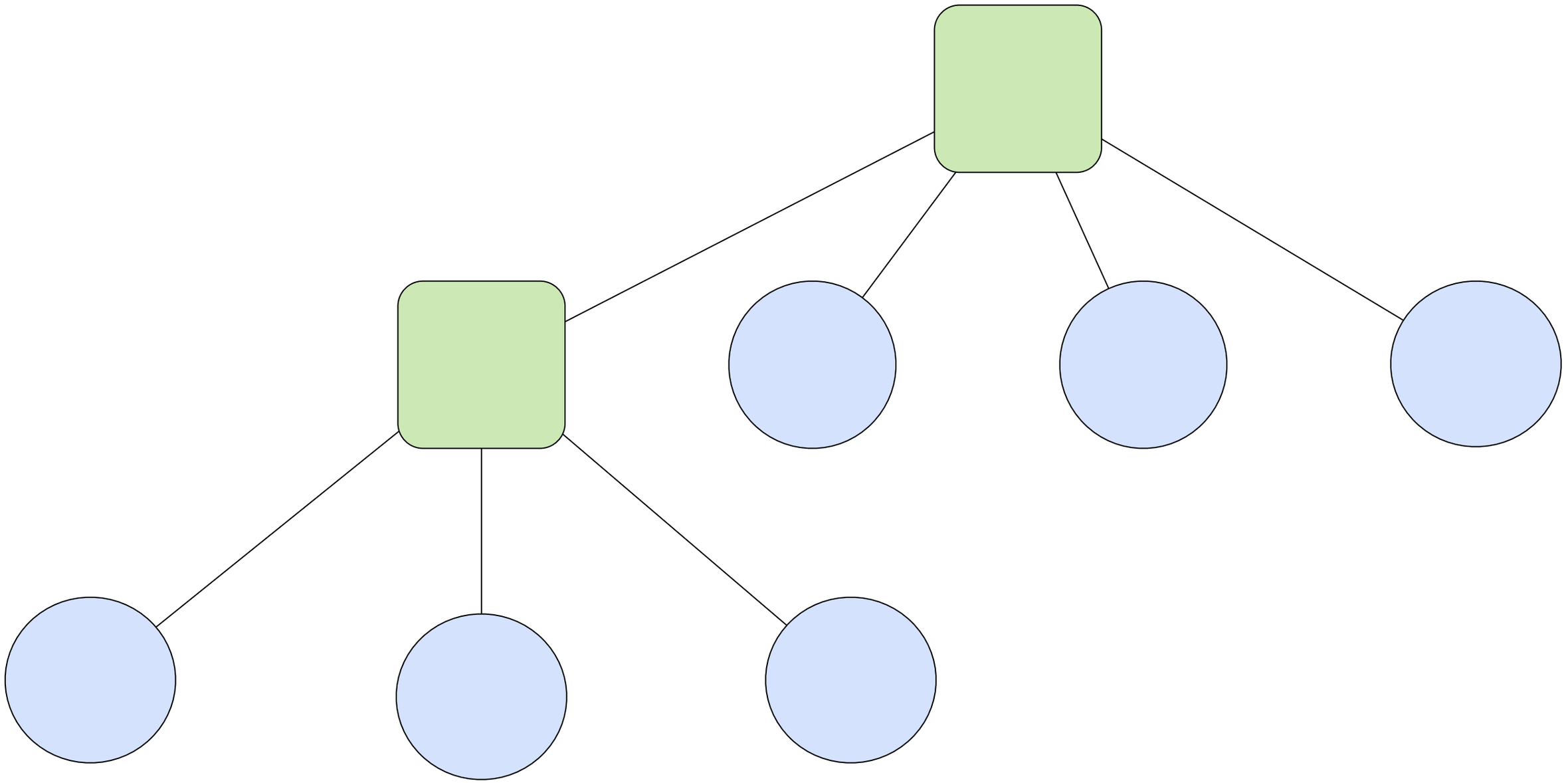
Supervisor hierarchies



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
  }
}
```

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”, new 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)
```

STM

another tool in the toolbox

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

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]()
```

STM: API

```
import Transaction.Local._

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

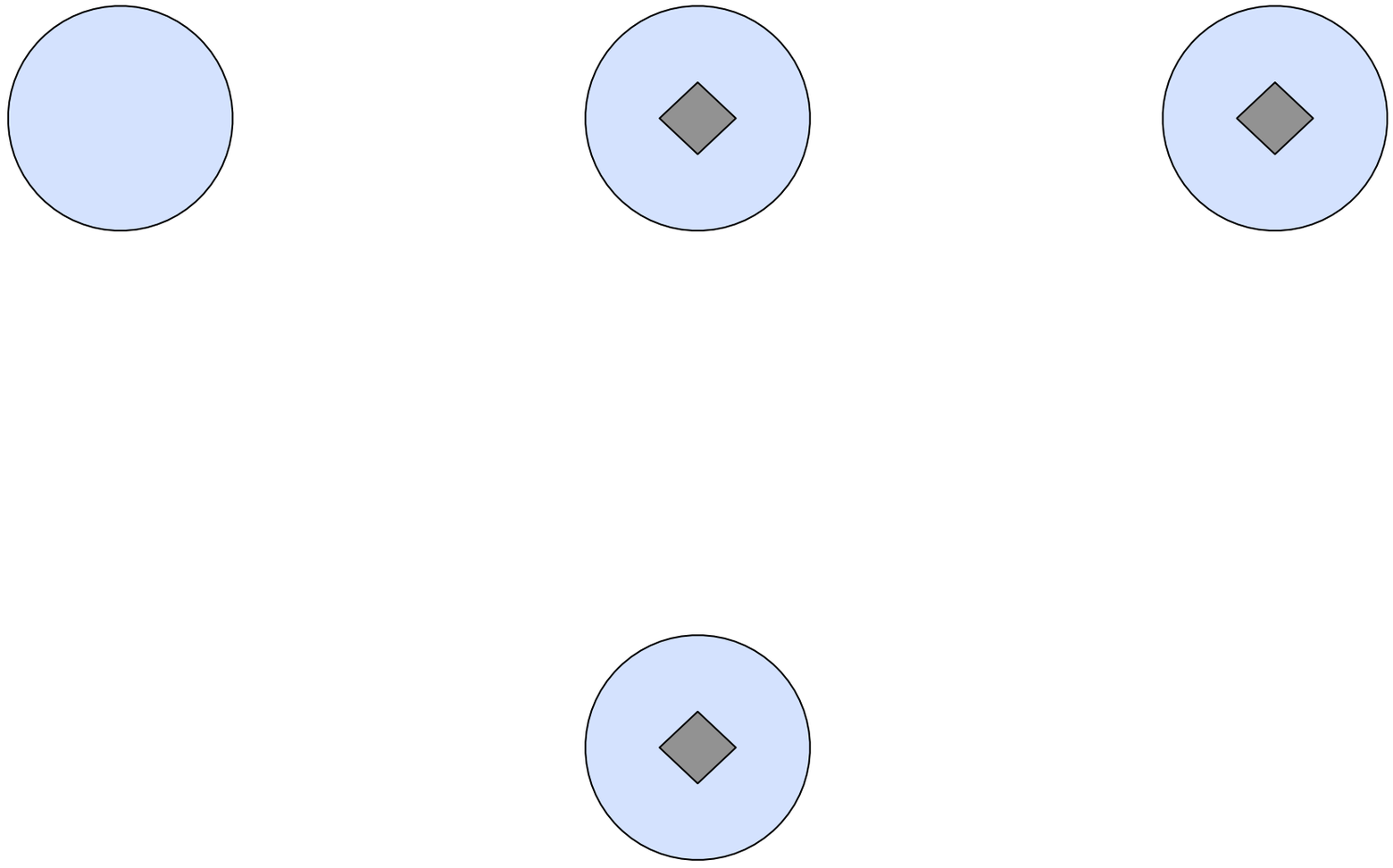
Actors + STM =

Transactors

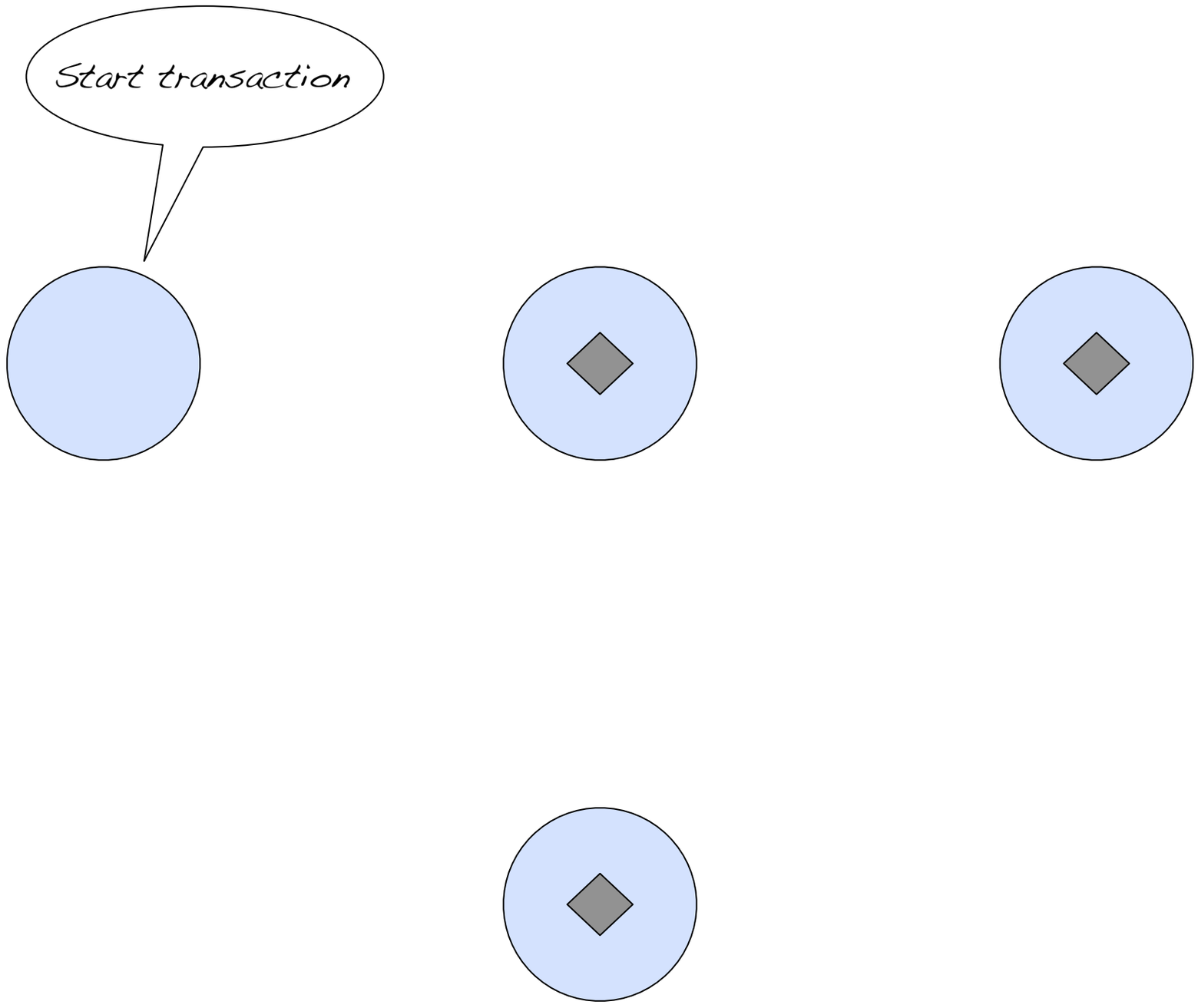
Transactors

```
class UserRegistry extends Transactor {

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

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

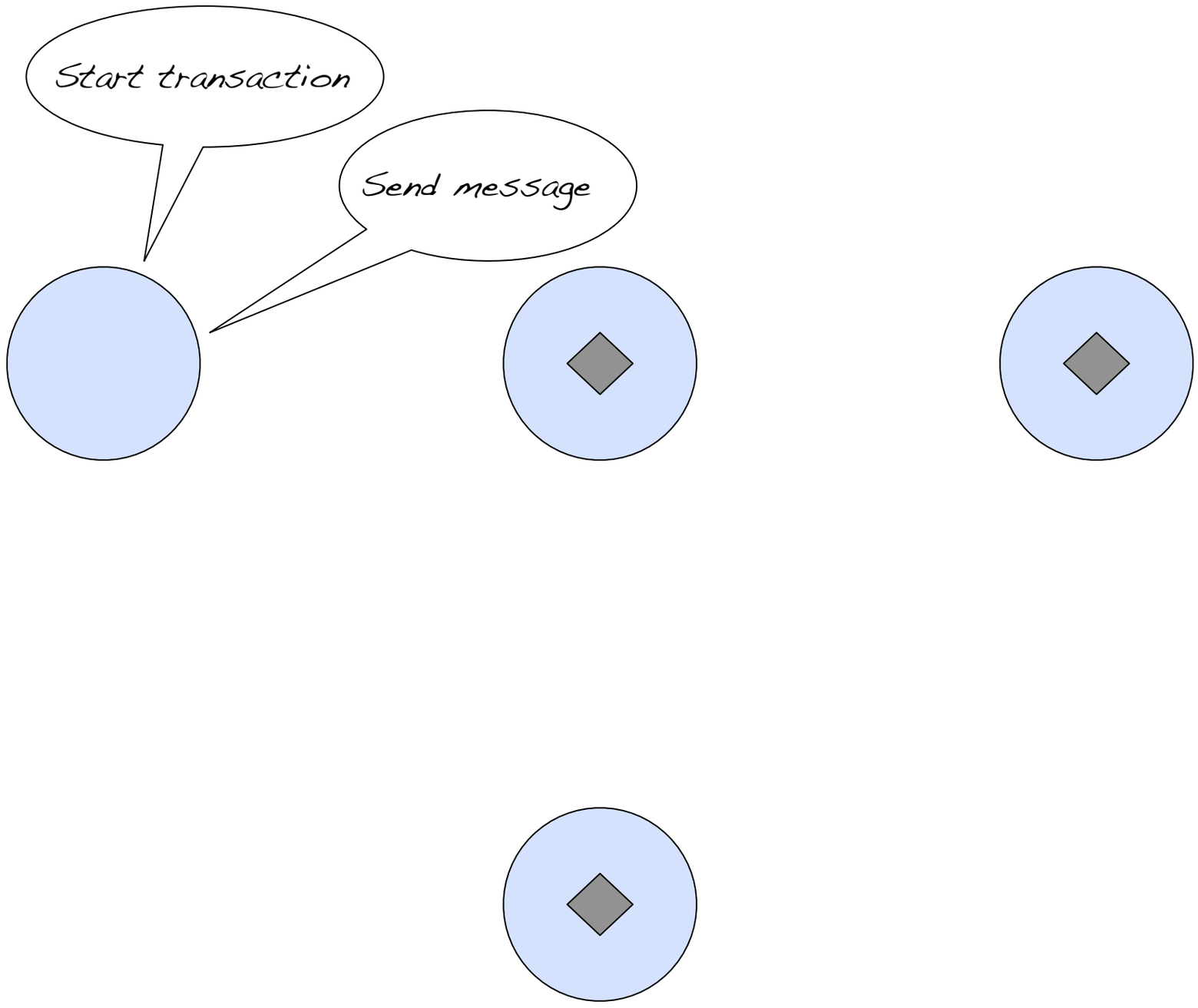
Transactors



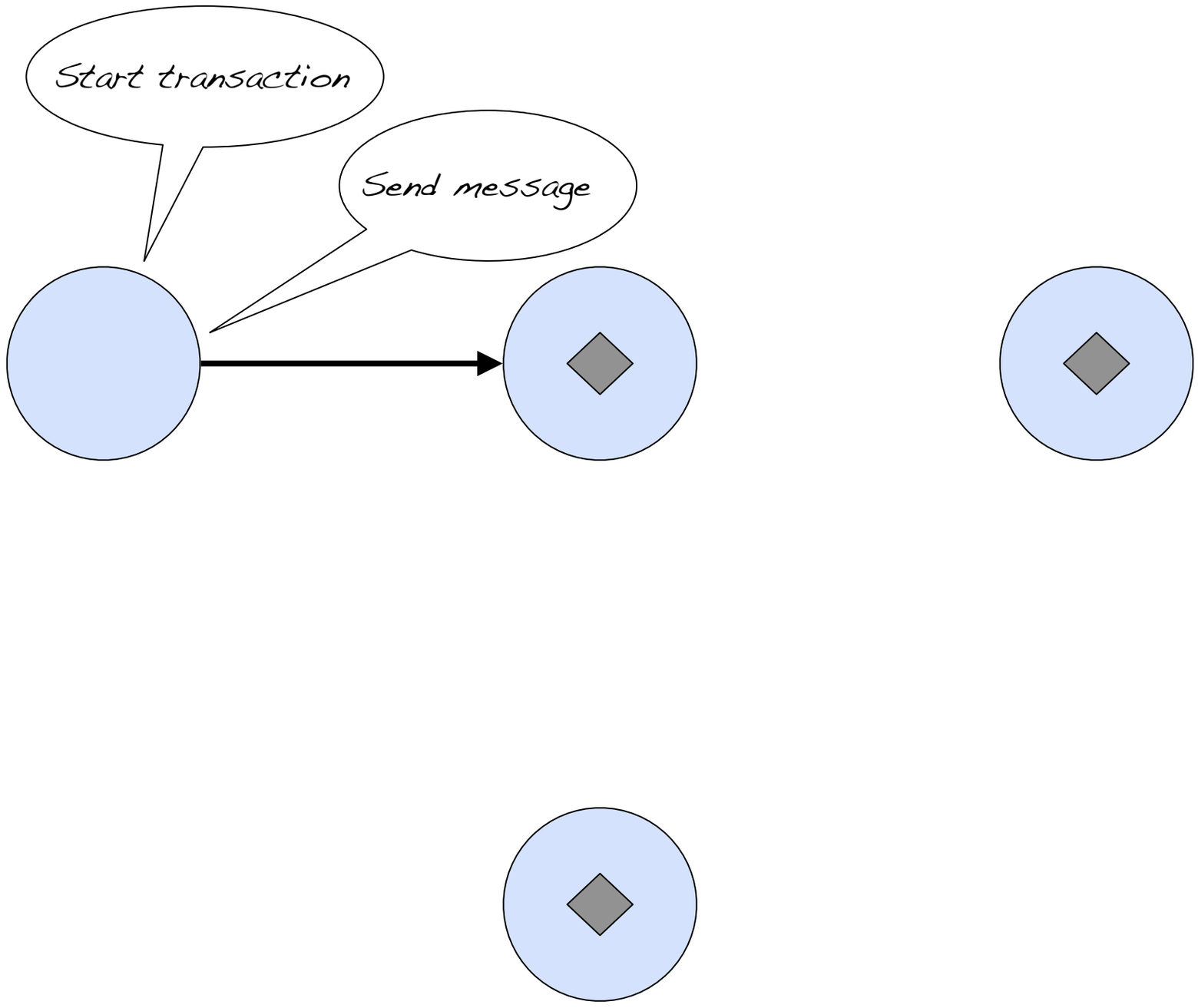
Transactors



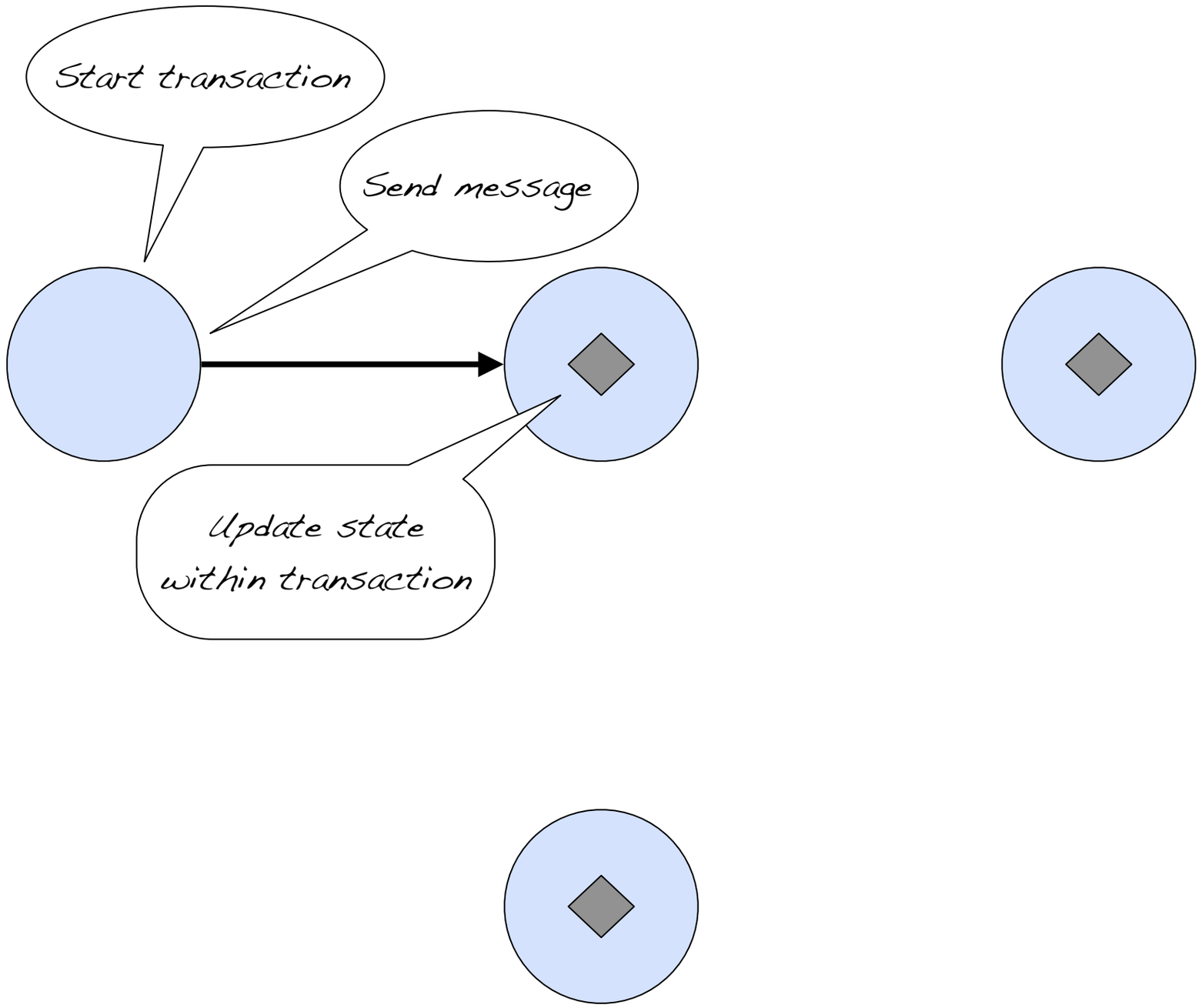
Transactors



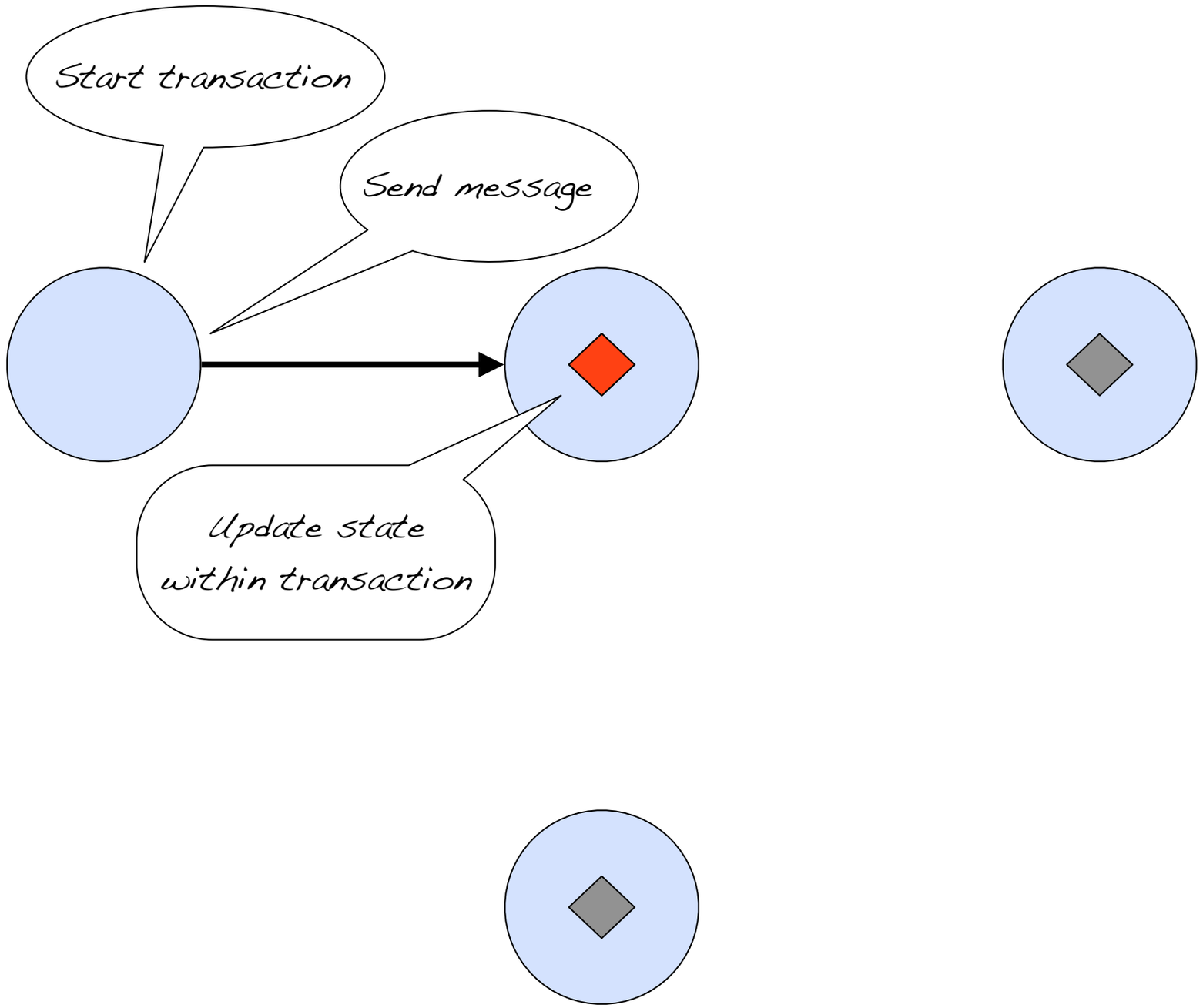
Transactors



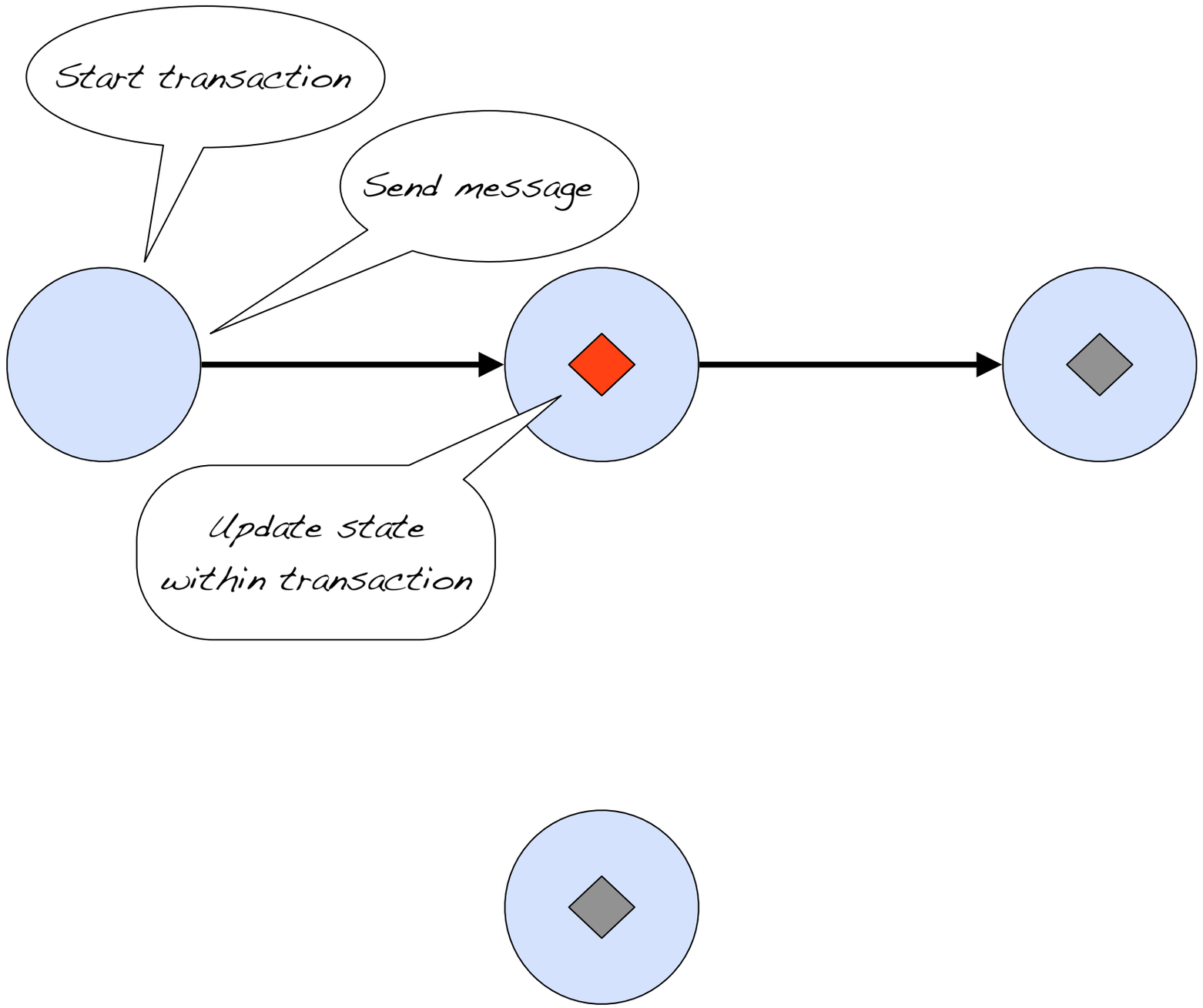
Transactors



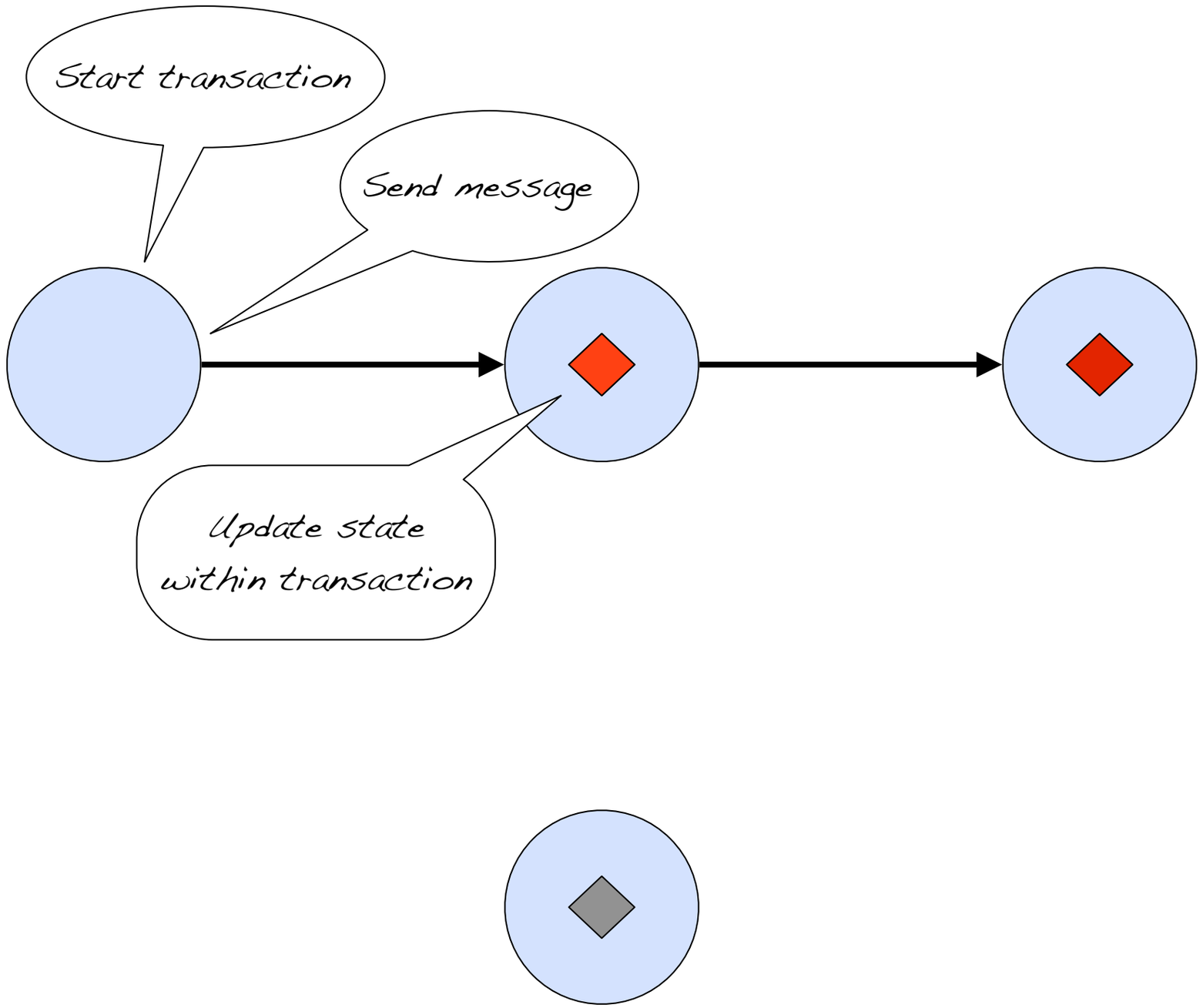
Transactors



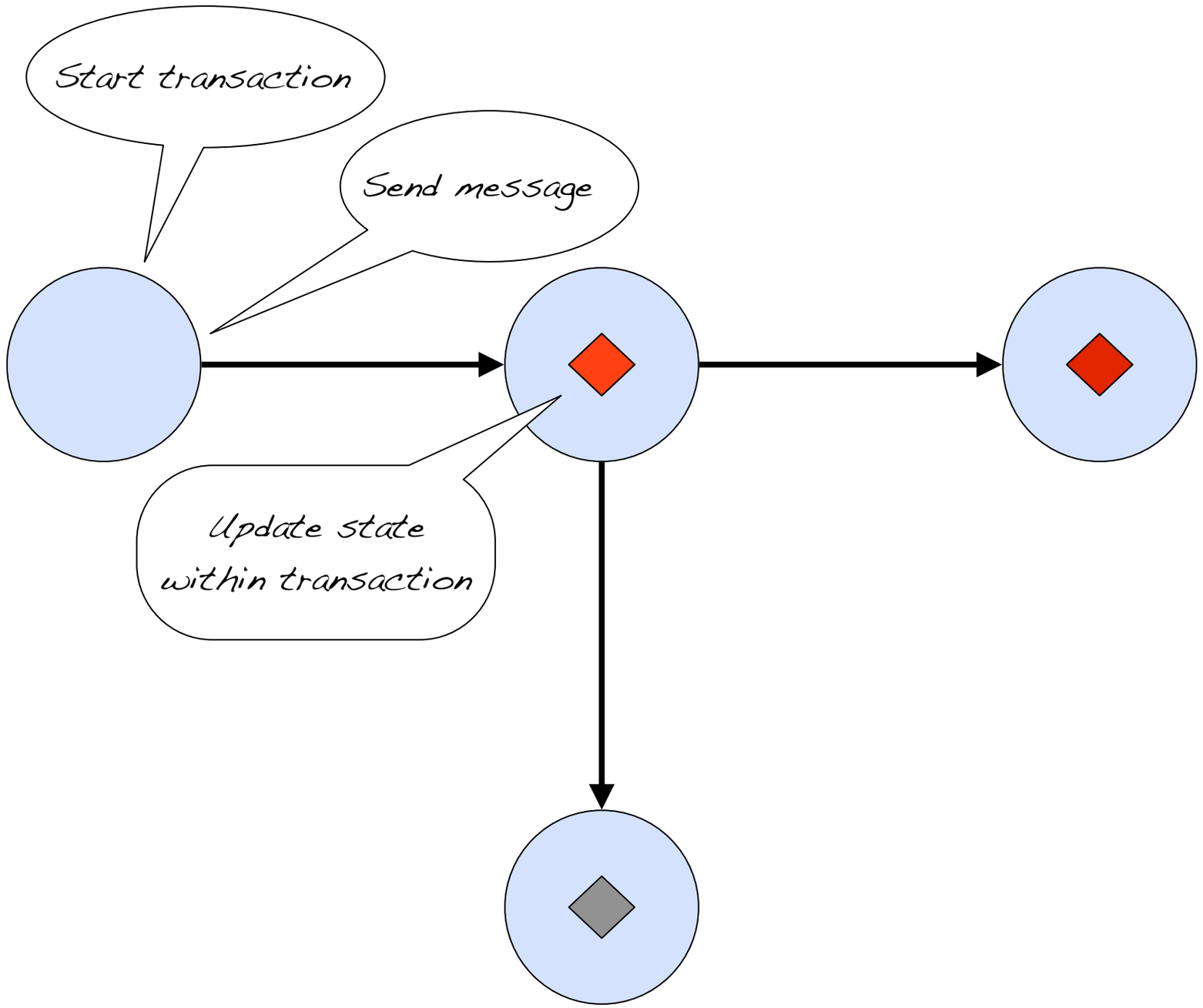
Transactors



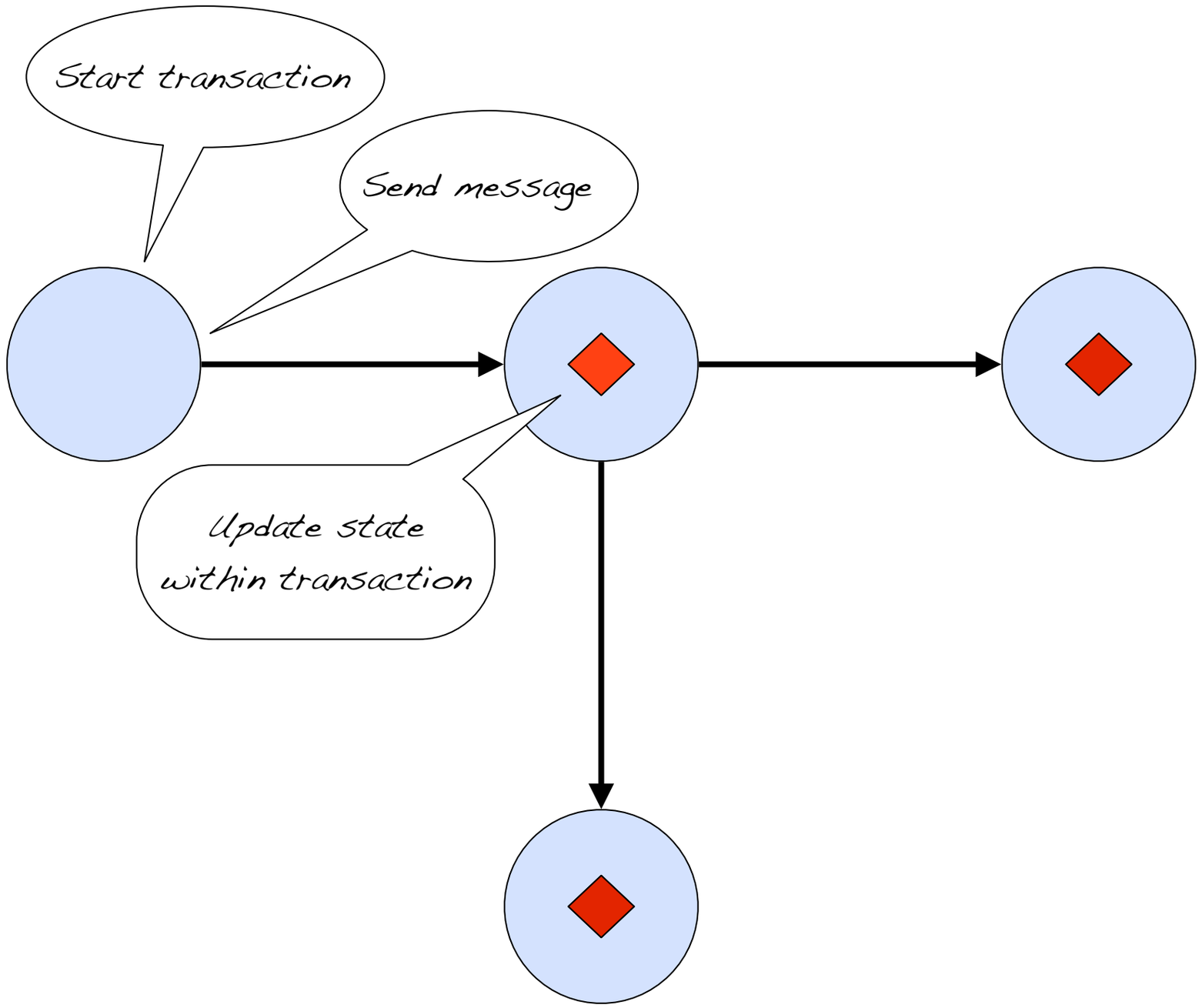
Transactors



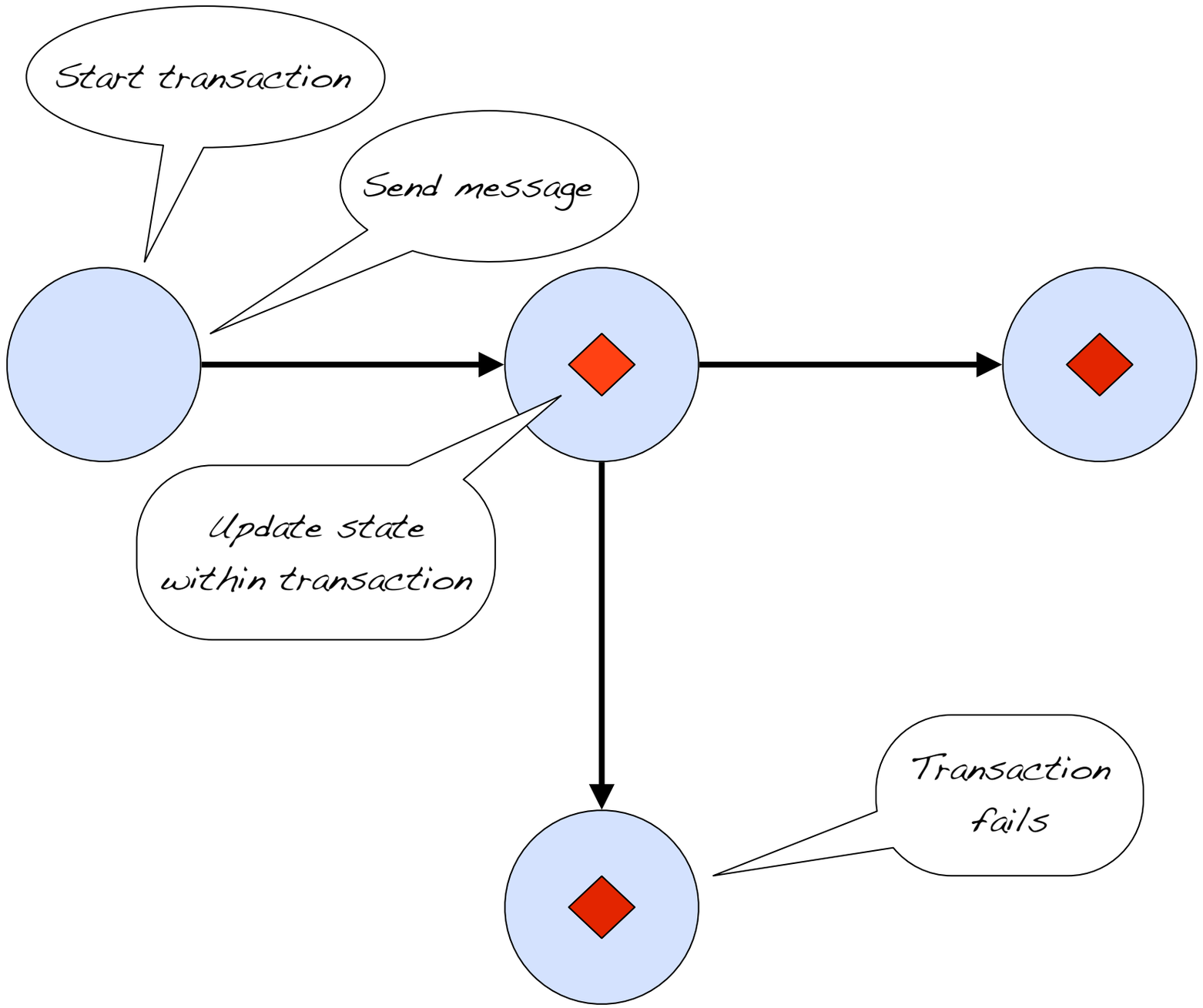
Transactors



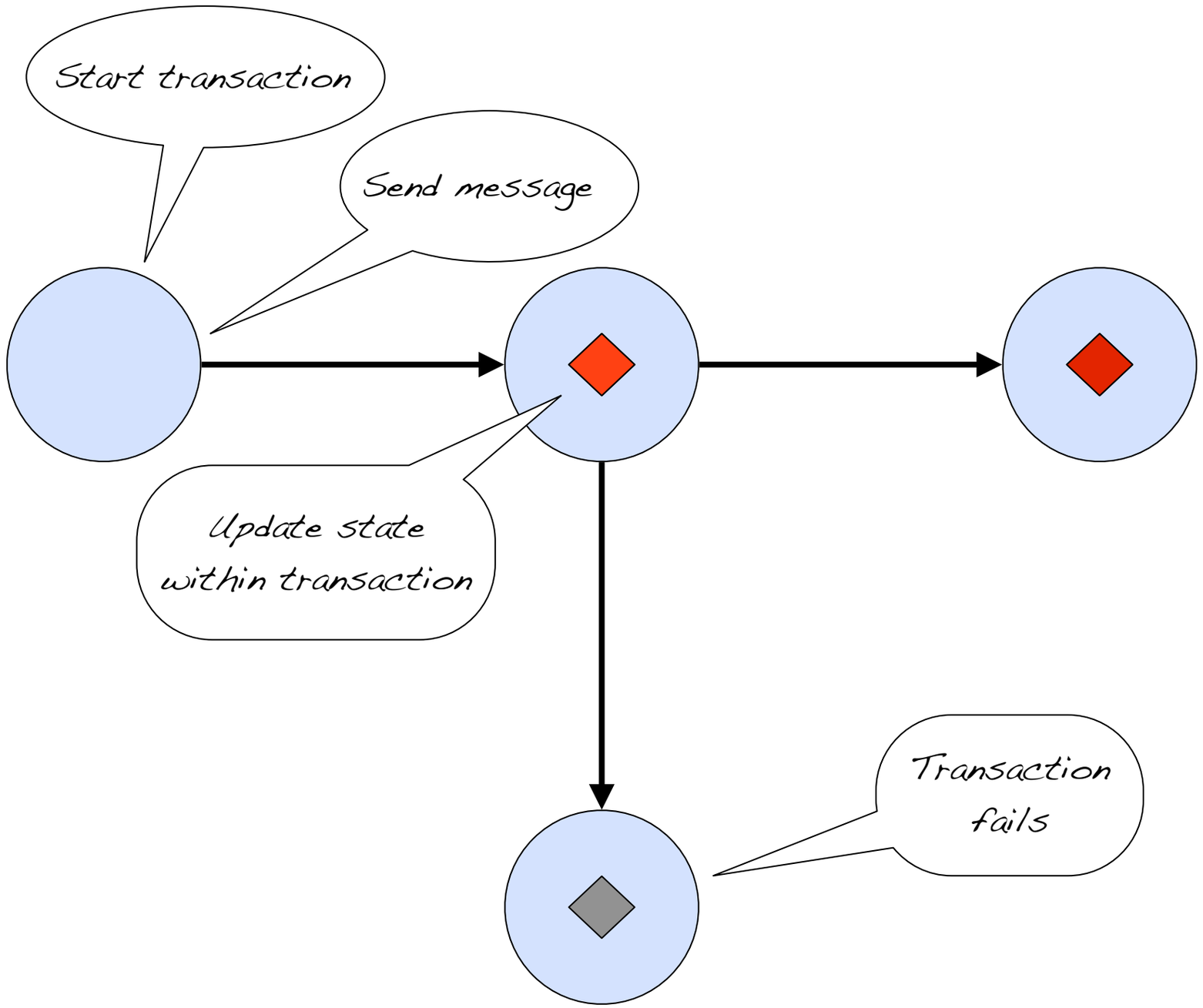
Transactors



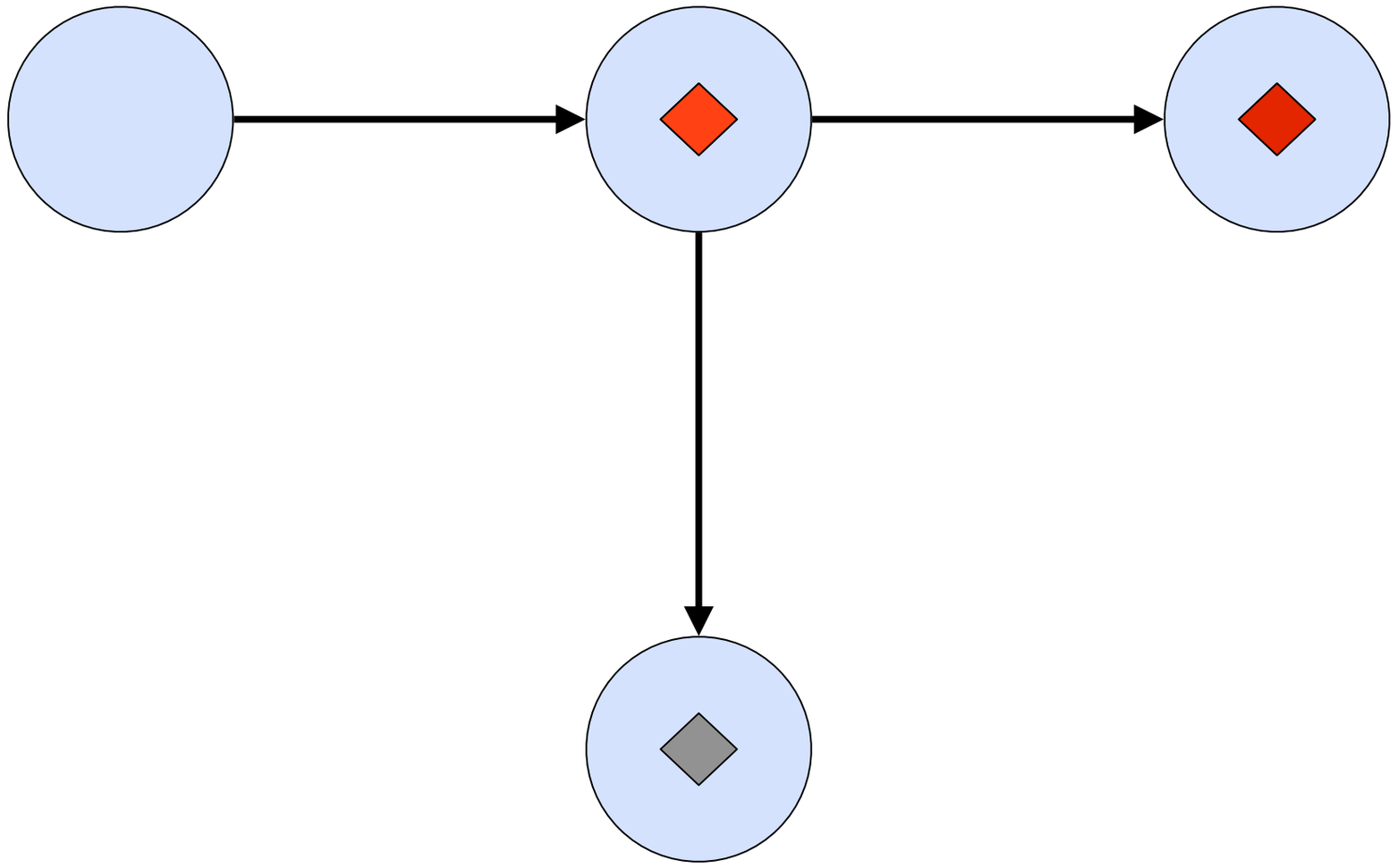
Transactors



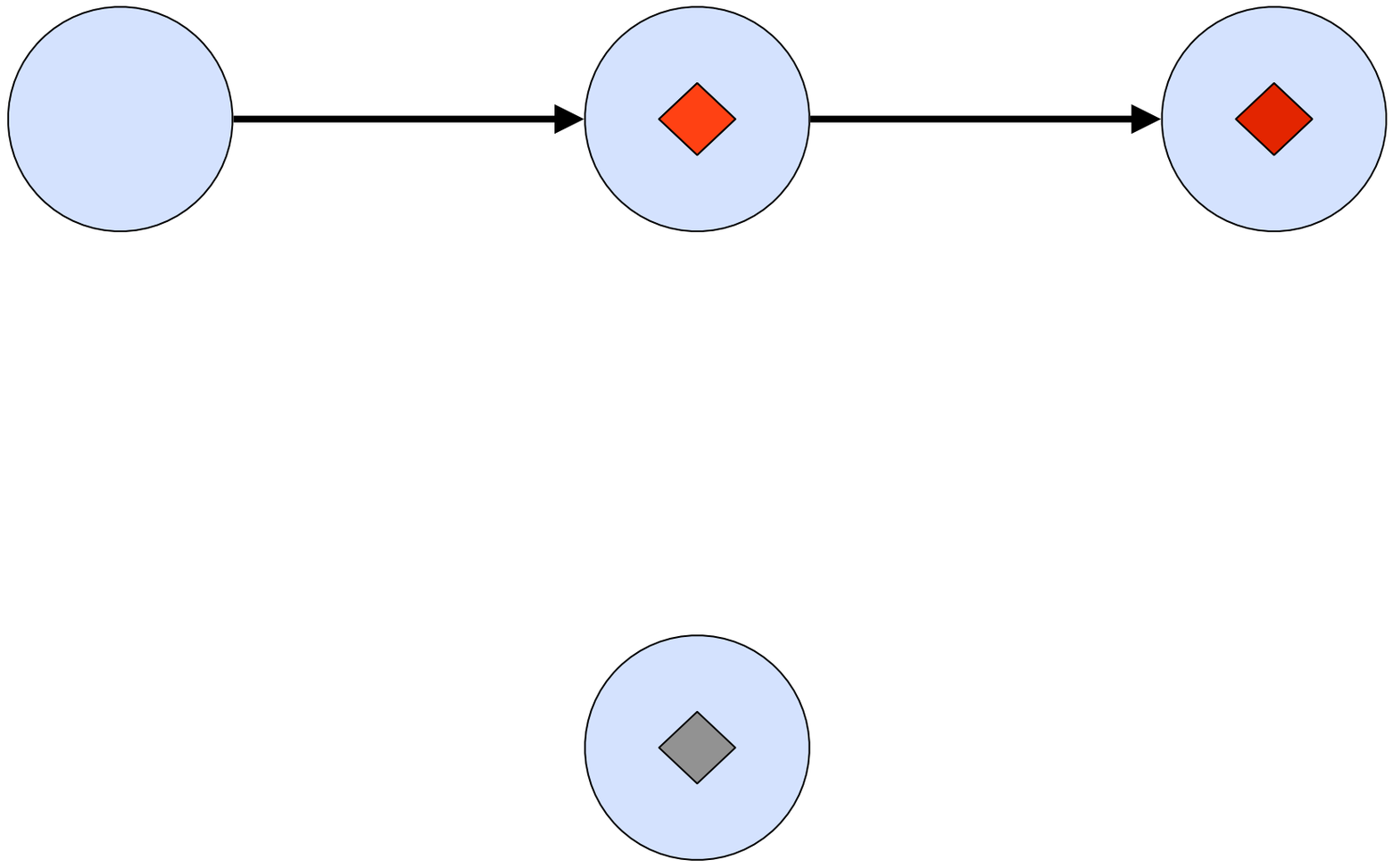
Transactors



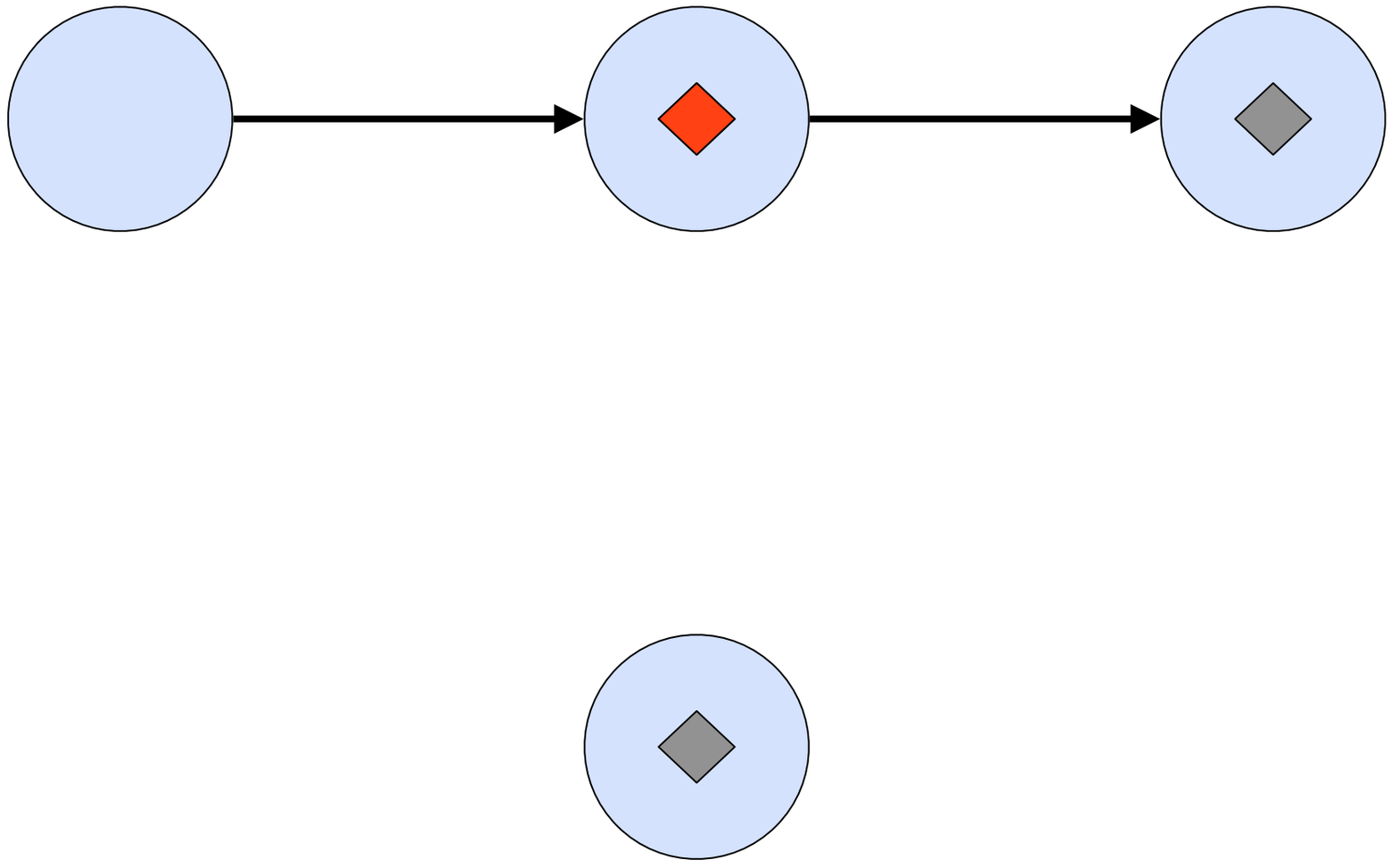
Transactors



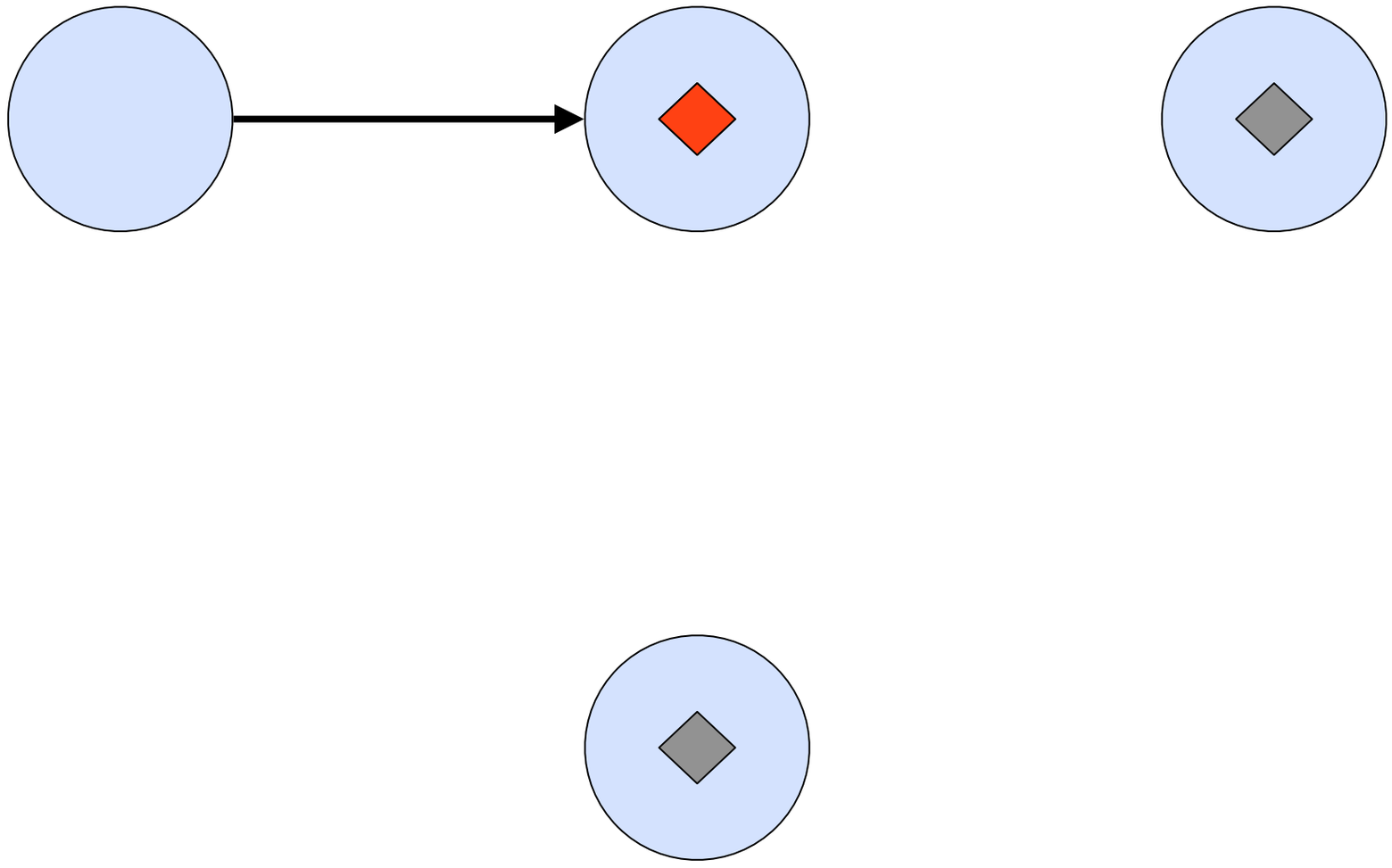
Transactors



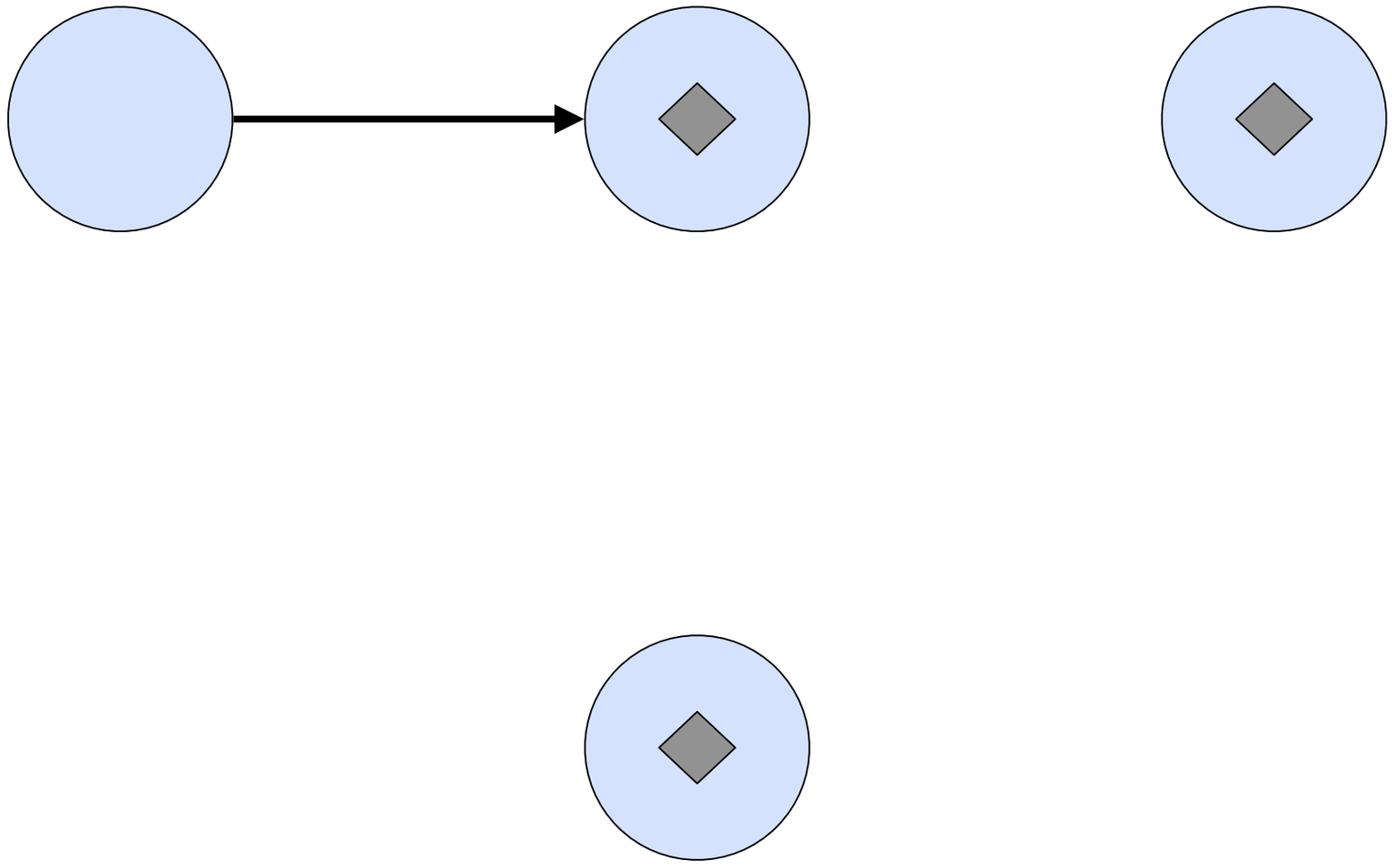
Transactors



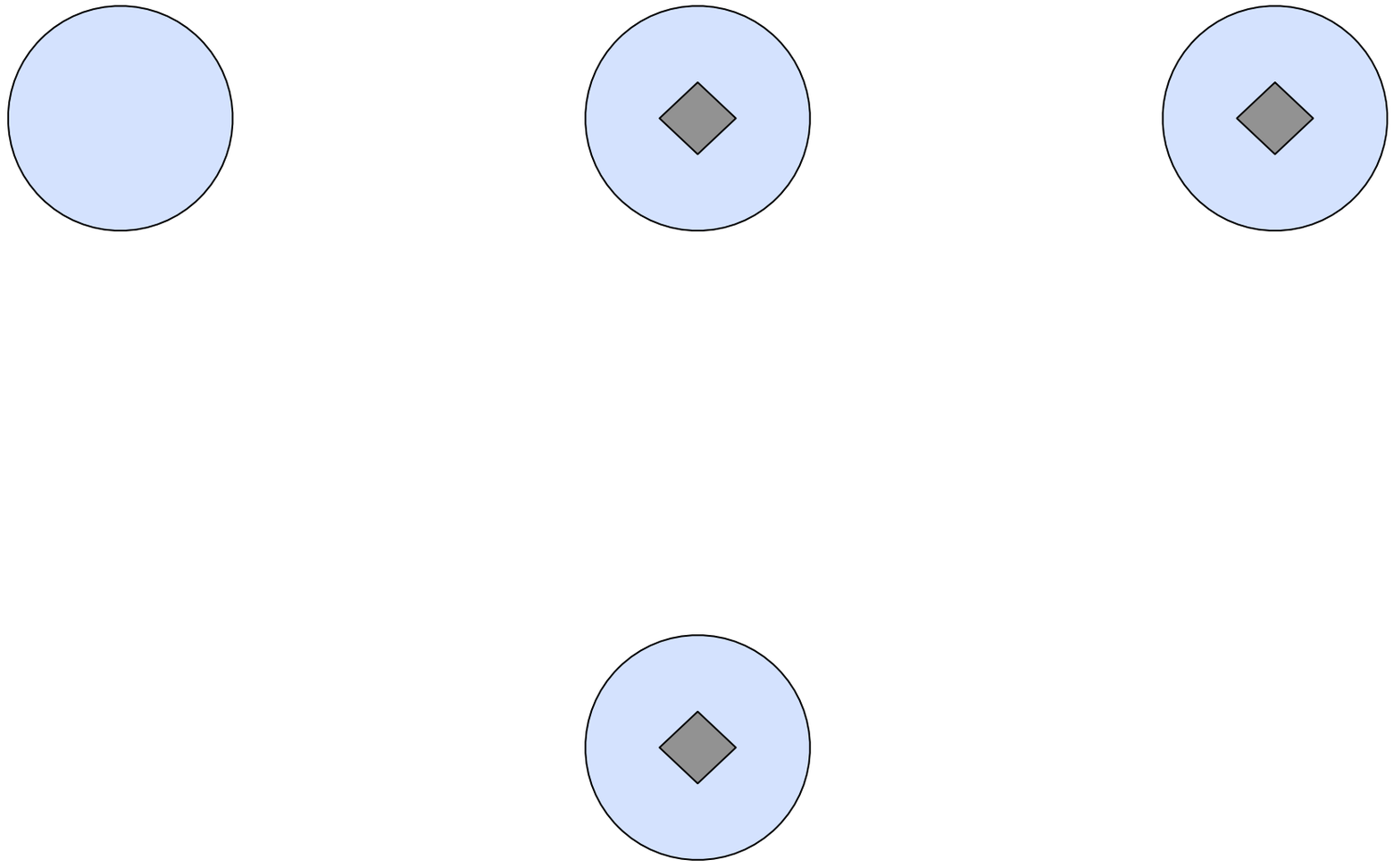
Transactors



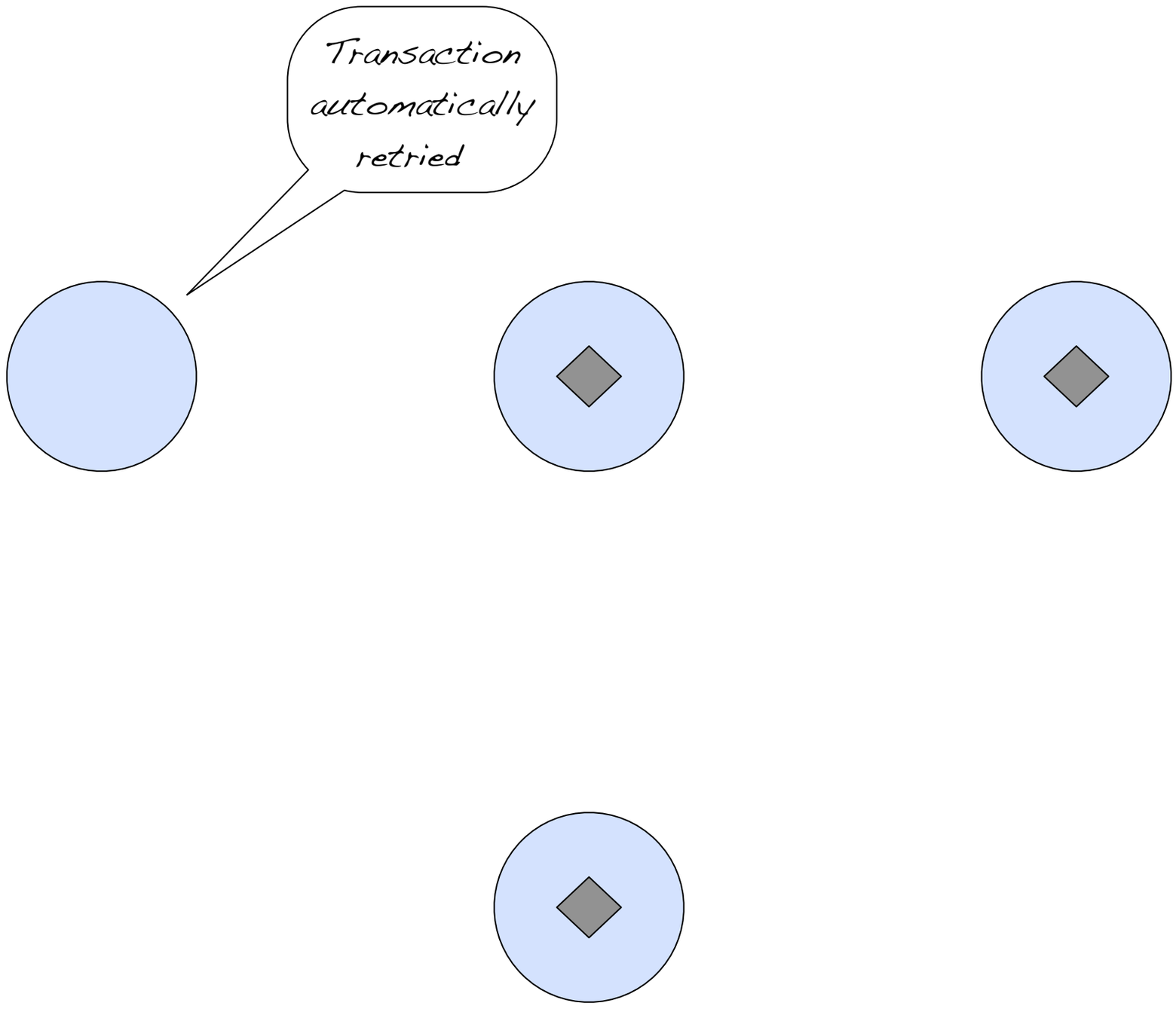
Transactors



Transactors



Transactors



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

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 Persistence

Persistence

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

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

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

...and much much more

Camel

REST

Security

Spring

Web

Comet

Guice

AMQP

Learn more

http://akkasource.org

akka source

The commercial entity behind Akka

http://akkasource.com

EOF

?