

Transactors

Simpler Concurrency

Transactors

Simpler Concurrency

VIKTOR KLANG
R&D DIRECTOR



<http://github.com/viktorklang/transactors>

The Short Story

- Developer since 1998
- Started on C
- Been on the JVM since 2001
- Background in business software
- Pragmatist
- OSS Enthusiast
- Concurrency Maniac

About Akka

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

Before we start

- This will be **JVM**-focused
- There is **no** general optimal solution
- Problems, tools and the problem with tools
- I expect a **continuous flow** of questions!

This presentation will hurt

But it **will** be worth it

What is Concurrency?

What about Parallelism?

Computer programs
are **lifeless** collections
of instructions

Processes

A Process is

- An **instance** of a computer program
- Has its own **timeline**
- Call **Stack**
- Address space (**Heap**)
- Security attributes and more...

Process gotchas

- OS Processes can be expensive to create (win32)
- Inter-Process Communication has overhead
- Context switching costs
- Hangs onto resources when idling
- Error management and recovery is hard

Threads

A Thread is

- An standalone execution path within a process
- Has its own **timeline**
- Call **Stack**
- Shares Address space (**Heap**) with other Threads within the same process

Thread gotchas

- Expensive to create
- Context switching costs
- Suffers from diminishing returns
- Hangs onto resources when idling
- Lifecycle is hard to control from the outside
- Error handling and recovery is hard

Others

- **Actors**
- Fibers
- Executor services
- Dataflow concurrency
- Parallel collections
- Agents
- and more...

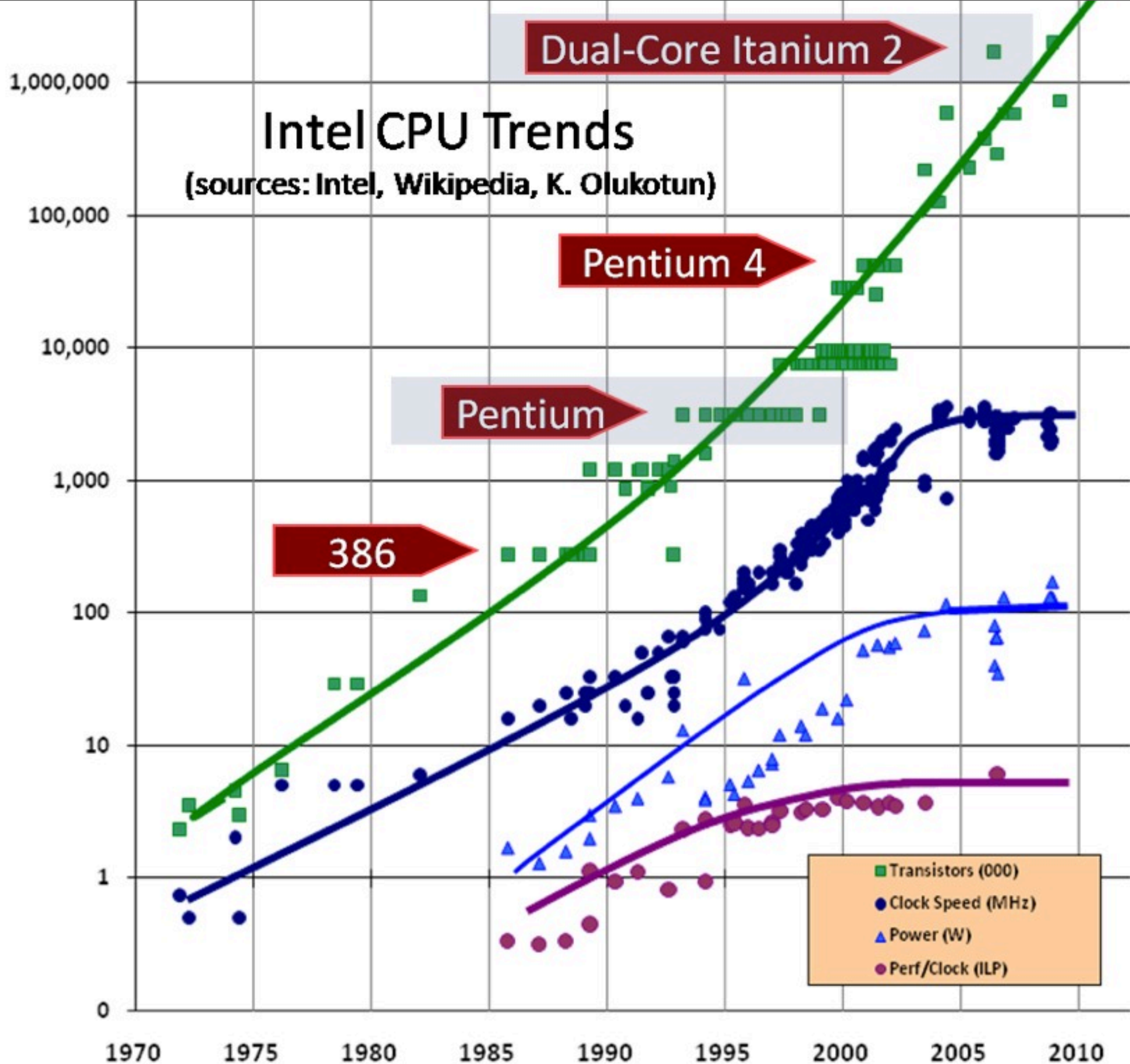
Schedulers

- The job of the **OS** scheduler is to allocate available CPU time **between processes**
- The **JVM** has its own scheduler to distribute its share of CPU time between threads

Some
bad news guys...

The free lunch is over

Herb Sutter 2005 <http://www.gotw.ca/publications/concurrency-ddj.htm>



Chip manufacturers will
focus on multicore technology
as the way to
improve performance

As a result -

Software developers will be
forced to write massively
multithreaded software

Some

good news...

Multicore

gives us

parallelism

Sync vs Async

The SLA

- 4 Services
 1. 3 seconds
 2. 4 seconds
 3. 1 second
 4. 6 seconds
- The services are unrelated
- Results need to be joined and returned
- Your budget is 7 seconds

The SLA: Sync

```
def callServices = {  
  val first    = callService1 //3 seconds  
  val second   = callService2 //4 seconds  
  val third    = callService3 //1 seconds  
  val fourth   = callService4 //6 seconds  
  return first + second + third + fourth  
}
```

The SLA: Async

```
def callServices = {  
    val first    = callService1 //3 seconds  
    val second  = callService2 //4 seconds  
    val third    = callService3 //1 seconds  
    val fourth  = callService4 //6 seconds  
    return first.await + second.await +  
           third.await + fourth.await  
}
```

Result: The SLA

- Synchronous calls

14 seconds

- Asynchronous calls

6 seconds

The Store



Rules!

- Current stock is non-negative
- Must not allow overselling
- Must not miscommunicate to the customer
- More than 1 customer needs to be able to simultaneously use the shop

It's a minefield out
there!

Example pitfalls

- Race-conditions
- Deadlocks
- Lock Convoys
- Starvation
 - Livelocks
- Data corruption
- Visibility and ordering issues

Race-conditions

REPL time!

Deadlocks

ThreadA:

lockA.lock

lockB.lock

ThreadB:

lockB.lock

lockA.lock

REPL time!

Lock Convoys

Starvation & Livelocks

Data corruption

ThreadA:
print("foo")

ThreadB:
print("bar")

Result:

fbaroo

REPL time!

Visibility and ordering

REPL time!

```
class SomeClass {  
    var first = 0  
    var second = 0  
  
    def foo() {  
        first = 1  
        second = 1  
    }  
  
    def bar() {  
        if (second == 1 && first == 0)  
            println("can't happen")  
    }  
}
```


The usual suspects

- Mutable state
- State dependencies
- Side effects
- Wrong use of concurrency control structures/techniques

```
class Counter {  
    private val count: Long = 1000000  
  
    def increment(): Long = count += 1  
  
    def decrement(): Long = count -= 1  
  
    def set(newCount: Long) {  
        count = newCount  
    }  
  
    def value(): Long = count  
}
```

Concurrency Control:

Encoding correct behavior

Concurrency primitives

Volatile variables

- Makes sure that reads and writes of the variable are against main memory
- Access synchronizes all cached copies of variables with main memory
- Reads and writes are **atomic, but NOT lockable.**

Monitor

- A thread safe? object
- **Serializes** execution of its methods
- Employs the use of **wait** and **notify**
- `java.lang.Object` is a Monitor but requires the use of the **synchronized** keyword to mark methods that needs mutual exclusion

Locks

- Controls access to a common resource
- May support reentrancy
- May support read/write

The loo



Photograph © [Andrew Dunn](#), 1992.

Semaphore

- Controls access to a common resource
- Binary (MUTual EXclusion) or Counting

The Restaurant



Latches

- A condition **starting out as false**
- When set to true **remains true forever**
- Enables Threads to **wait for it** to become true

Barrier

- Any thread must **stop at the barrier**
- Cannot proceed until **enough threads** have arrived
- Can be viewed as **a threshold**

Compare-And-Swap

- Atomic
- Conditional modification
- Low level construct
- Very useful

CTS example

```
class MyClass {  
  private var isActive = false  
  
  def start() {  
    if (!isActive) {  
      isActive = true  
      //Code to be run on start  
    }  
  }  
  
  def shutdown() {  
    if (isActive) {  
      isActive = false  
      //Code to be run on shutdown  
    }  
  }  
}
```

CAS example

```
class MyClass {  
  val isActive = new AtomicBoolean(false)  
  
  def start() {  
    if (isActive.compareAndSet(false,true)) {  
      //Code to be run on start  
    }  
  }  
  
  def shutdown() {  
    if (isActive.compareAndSet(true,false)) {  
      //Code to be run on shutdown  
    }  
  }  
}
```

Threads... well...

- The “Threads & locks”-combination is still taught as The Way to Do It
- The problem is
 - Knowing up front which locks need to be locked
 - Breaks encapsulation
 - All locks need to be taken before modification
 - In the correct order
 - All locks must be unlocked
 - Thread lifecycle management is error prone

“Threads are to Concurrency as
Witchcraft is to Physics”

“Hanging by a thread is the punishment
for Shared State Concurrency”

- Gilad Bracha

Java Language architect and maintainer and co-author of the Java Language Specification.

Software Transactional Memory (STM)

Transaction?

- A list of operations
- Atomic: All or nothing
- Usually
 - Atomic
 - Consistent
 - Isolated
 - Durable
- Example: Database transaction

STM Overview

- View 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** changes on abort

STM Overview

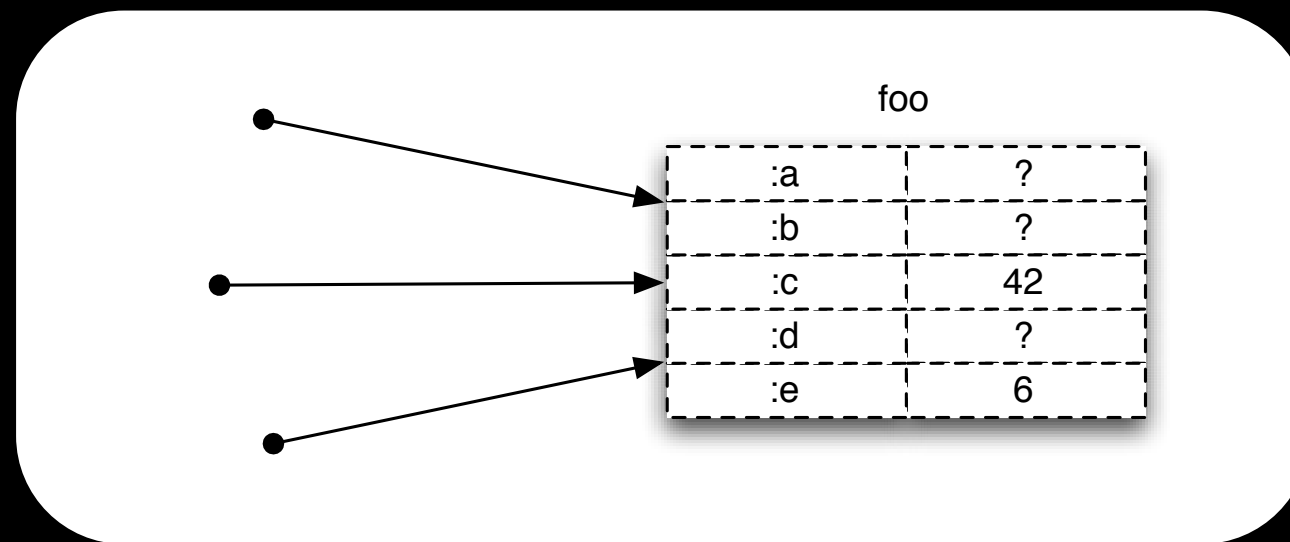
- Atomic
- Consistent
- Isolated
- Transactions can nest
- Transactions can compose!

STM restrictions

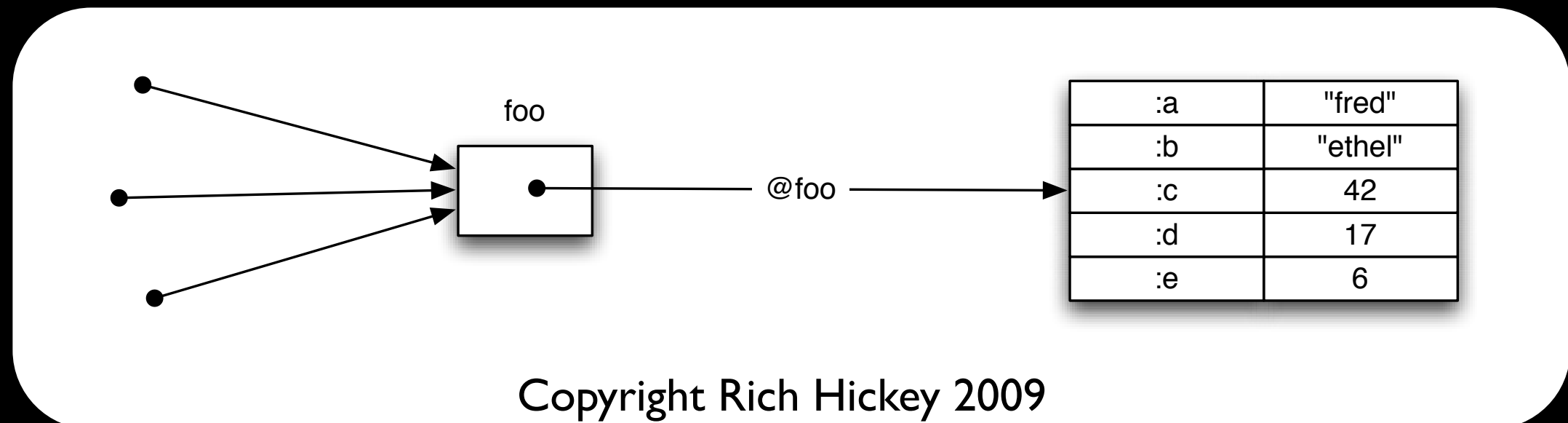
- Operations within the scope of a transaction
 - **need** to be idempotent
 - **can't** have side-effects

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

Transactional Reference

Creation:

```
val number = Ref(0)
```

Usage:

```
atomic {  
    number.alter( _ + 5 )  
    number.swap(10)  
    val result = number.get  
}
```

Transactional Map

Creation:

```
val beers = TransactionalMap[String,Float]( )
```

Usage:

```
atomic {  
    beers.put("Heineken",5.0f)  
    val alcoholPct = beers("Heineken")  
}
```

Transactional **Vector**

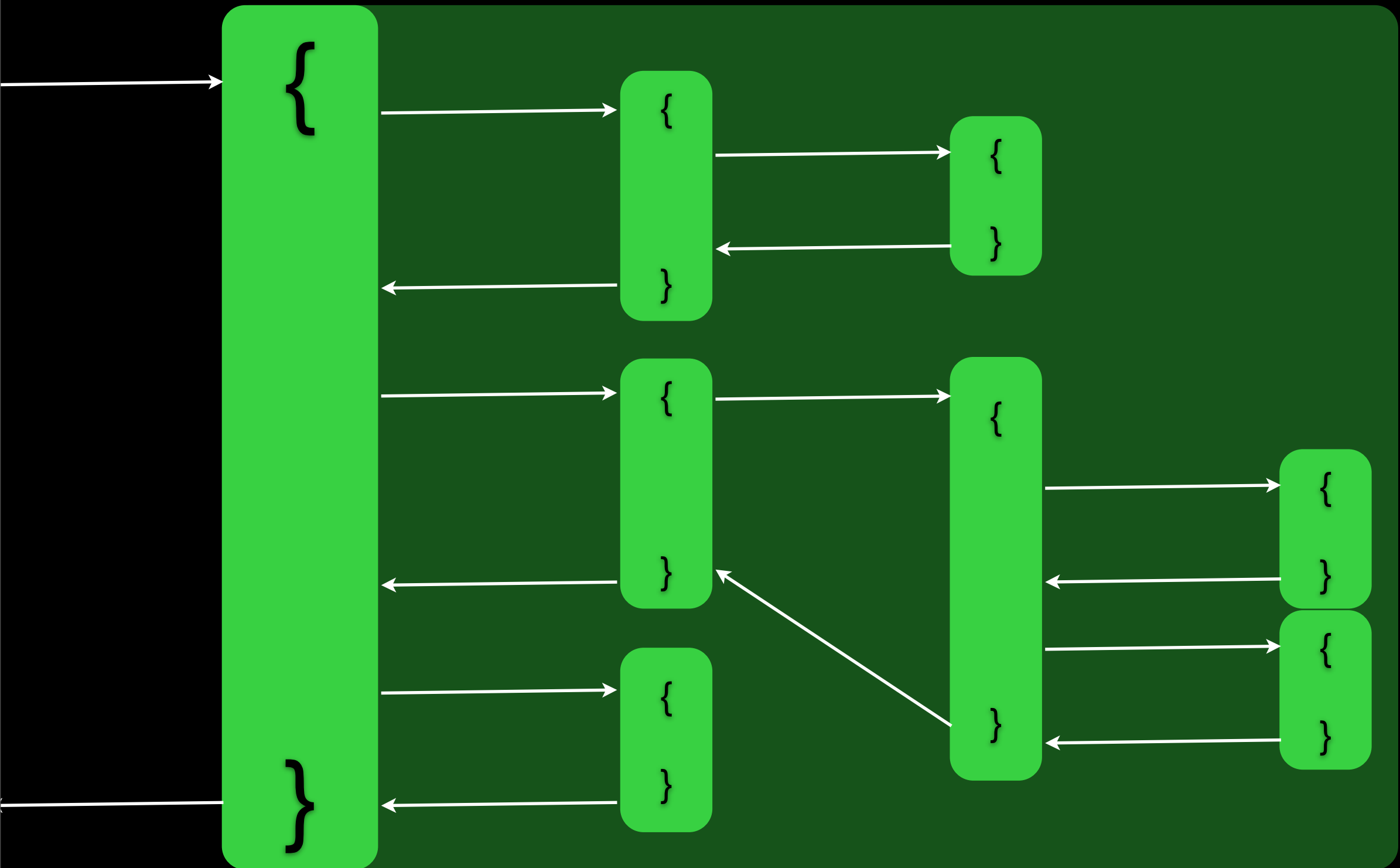
Creation:

```
val users = TransactionalVector(new User(""))
```

Usage:

```
atomic {  
    users.add(new User("Jonas"))  
    users.update(0, new User("Viktor"))  
    val first = users.get(0)  
}
```

Transaction composition

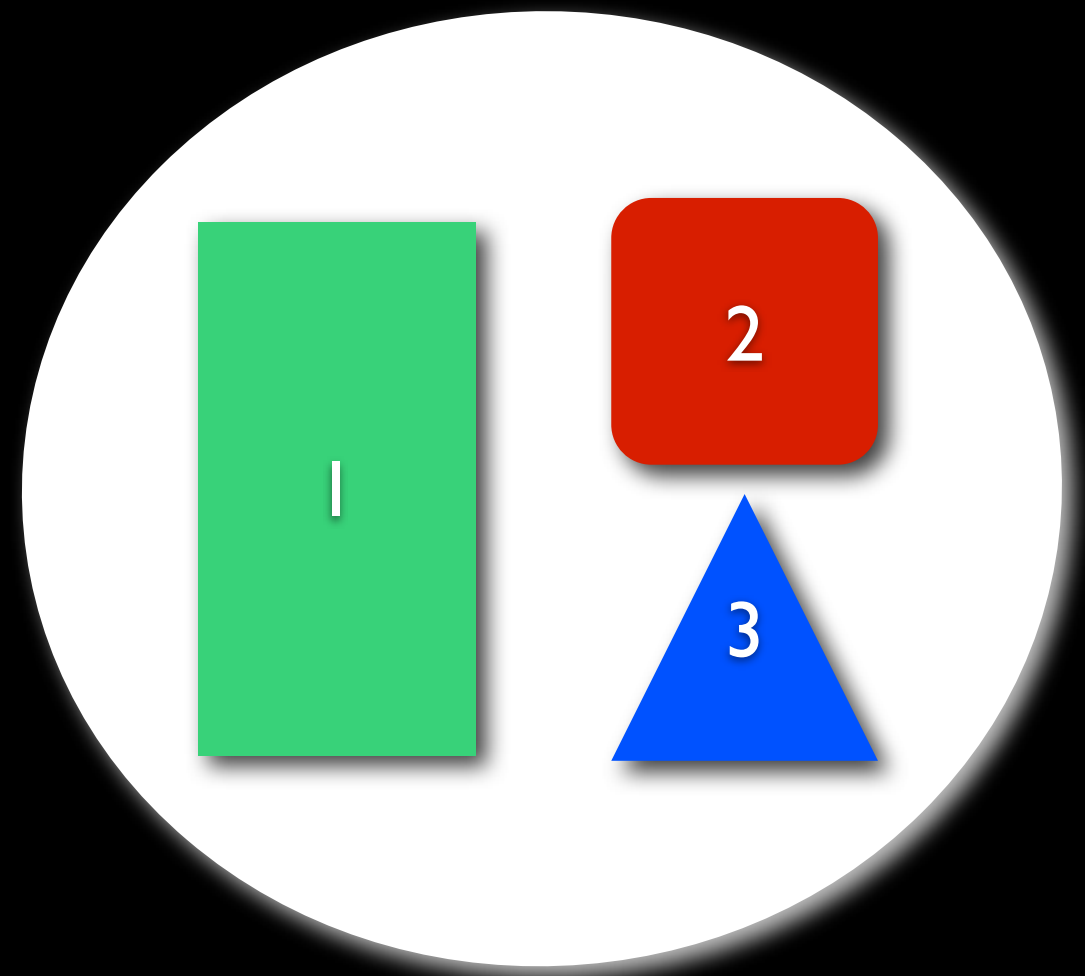


REPL time!

Actors

An Actor is

1. A mailbox
2. State (Optional)
3. Current behavior



Benefits of Actors

- One message at a time = no locks
- Inherently asynchronous
- Cheap!
- Literally MILLIONS at the same time

Actor Essentials

- An actor can send messages to other actors
- Can create new actors
- Can change behavior towards the next message
- Processes one message at a time

Lifecycle

- Unstarted
- Running
 - BeingRestarted
- Shutdown

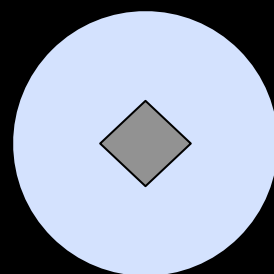
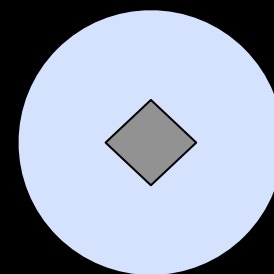
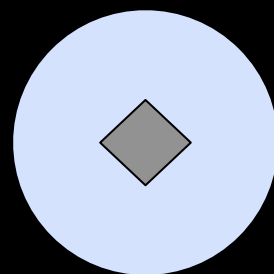
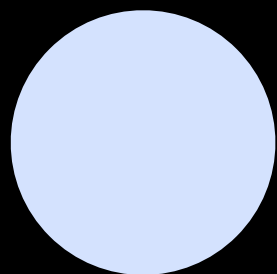
Defining an Actor

```
class MyActor extends Actor {  
  def receive = {  
    case "hello world" => reply("This is getting old man!")  
    case 5 => reply("I really don't like 5")  
    case i: Int => reply(i * i)  
    case _ => //I ignore other messages  
  }  
}
```

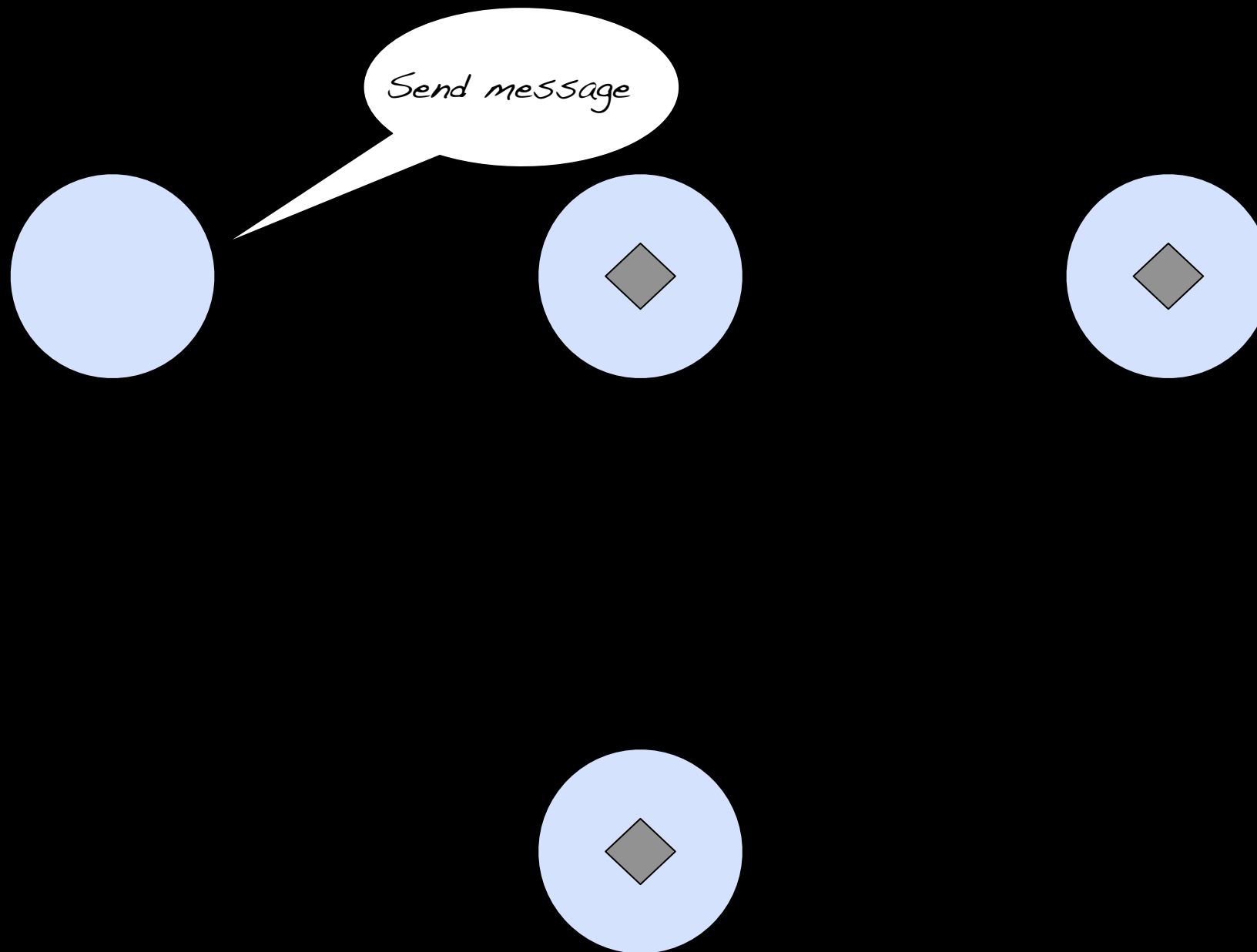
Sending messages

Method	Scala	Java
Send message	<code>actor ! message</code>	<code>actor.sendOneWay(message)</code>
Send reply within	<code>actor !! message</code>	<code>actor.sendRequestReply(message)</code>
Send reply eventually	<code>actor !!! message</code>	<code>actor.sendRequestReplyFuture(message)</code>

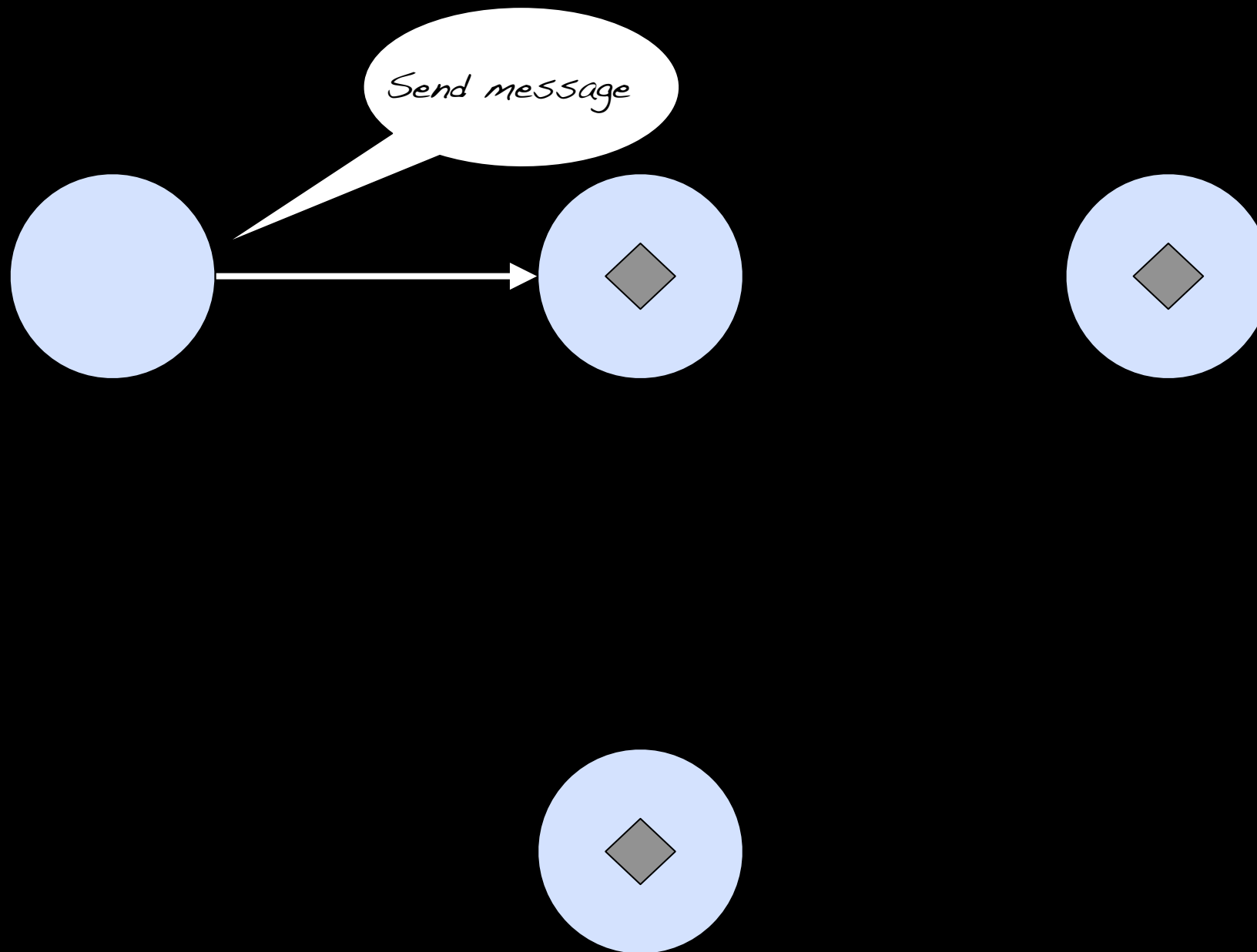
Actors



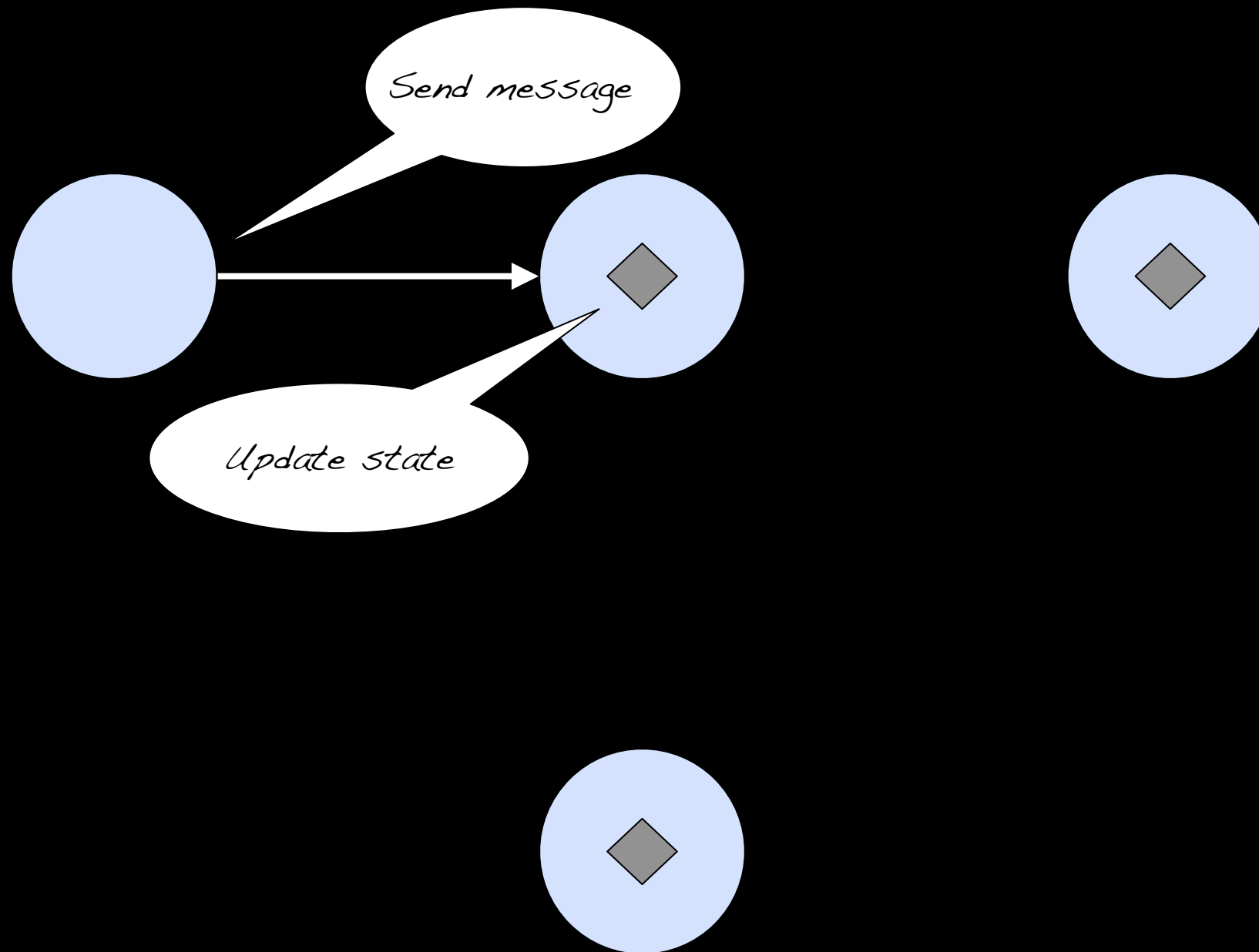
Actors



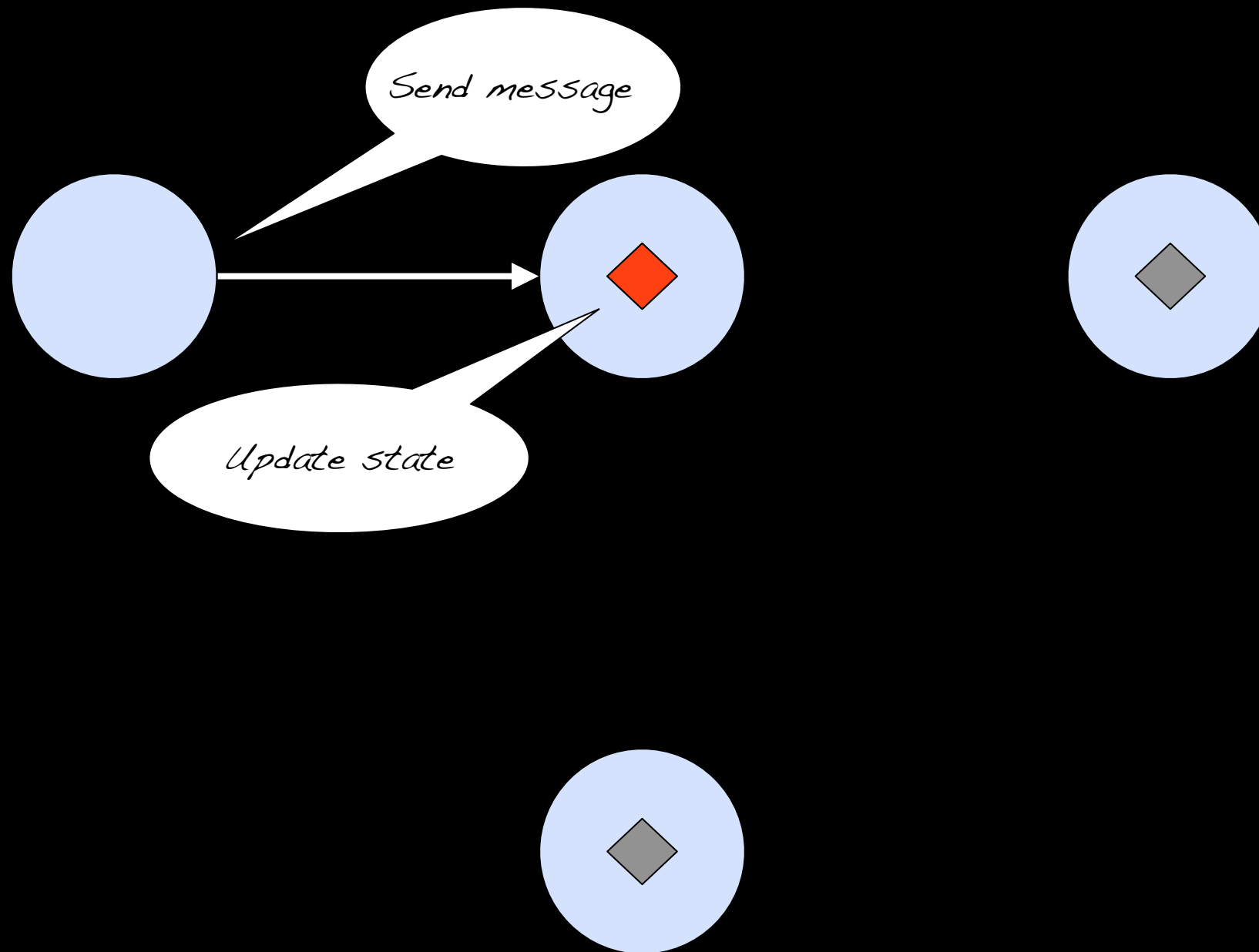
Actors



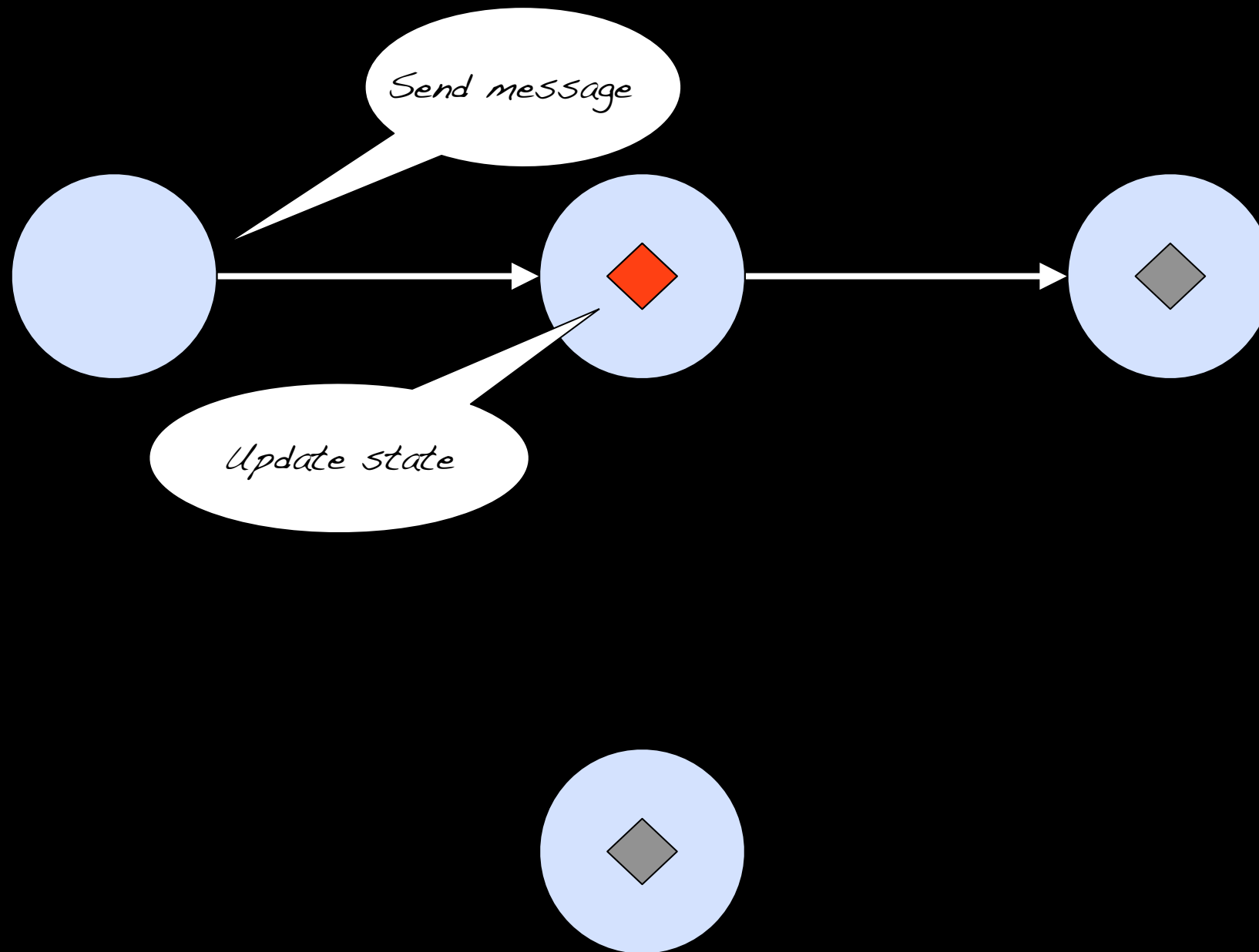
Actors



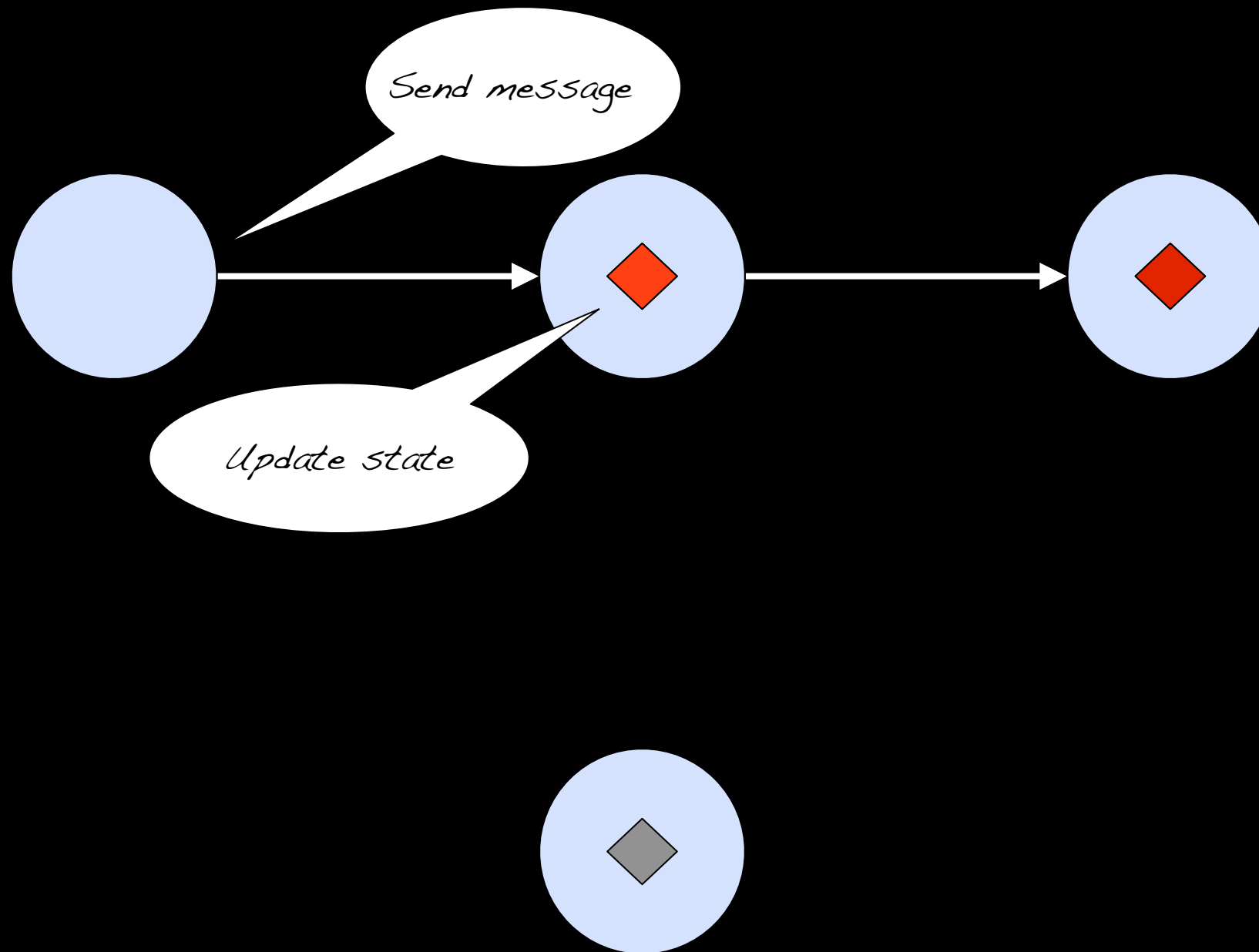
Actors



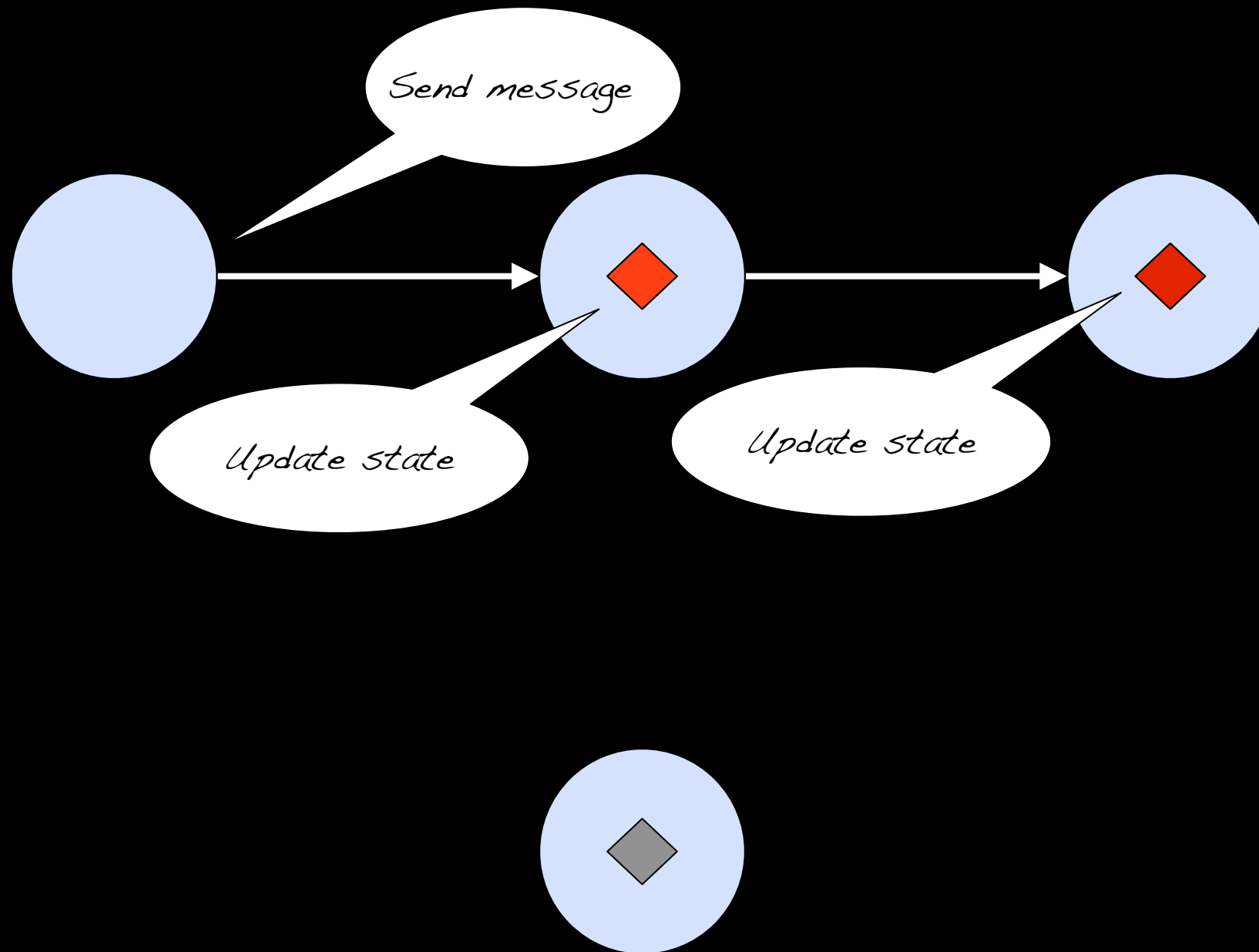
Actors



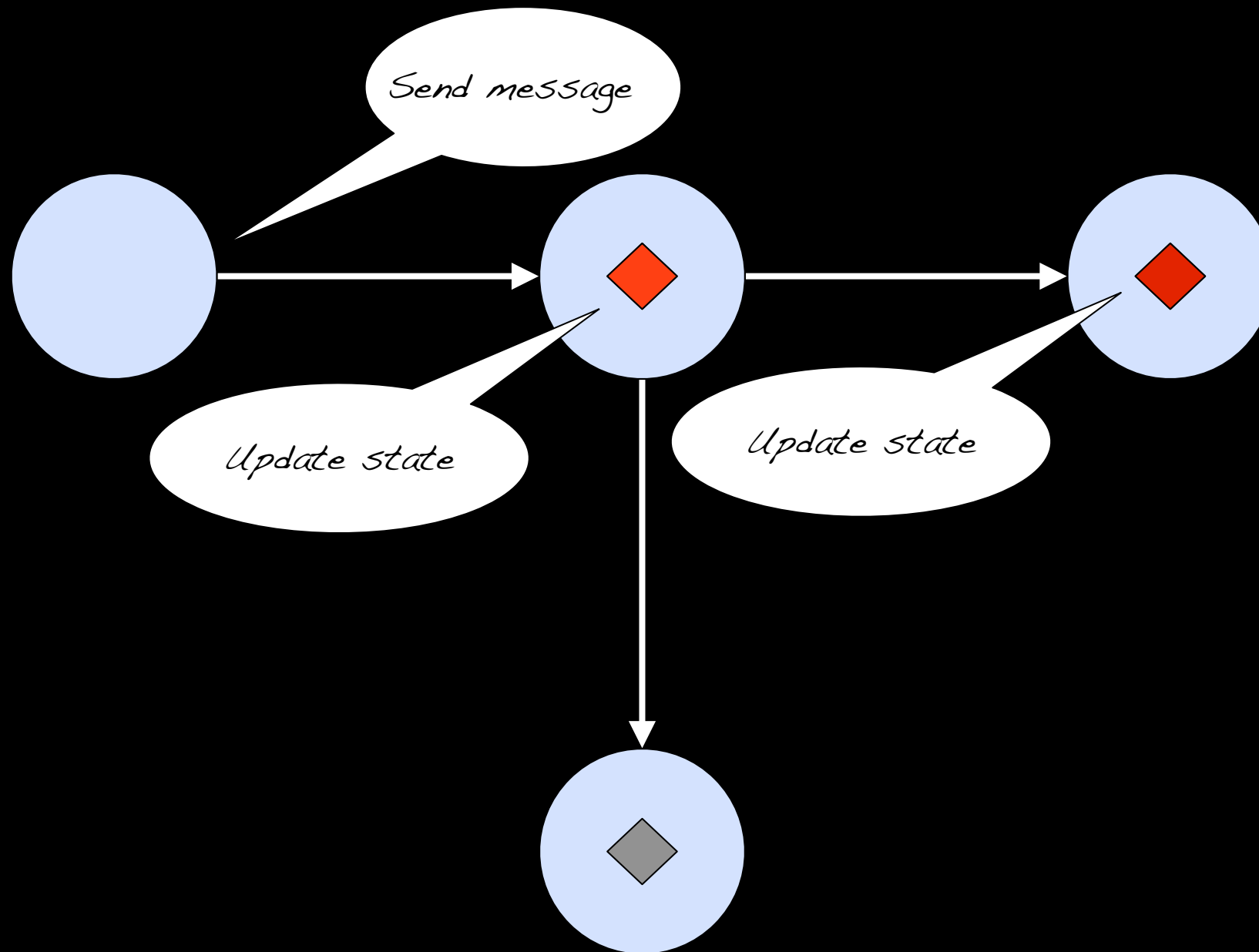
Actors



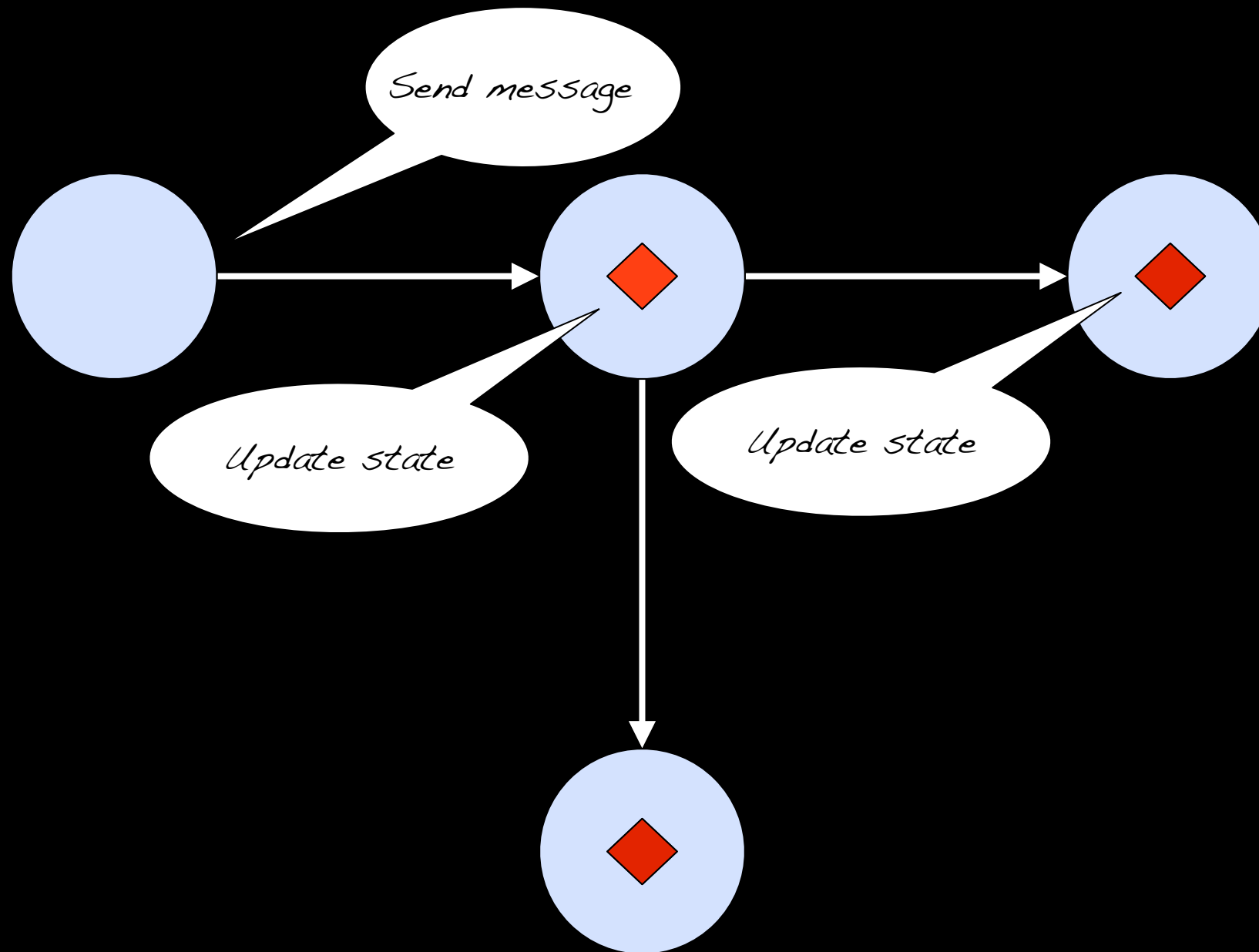
Actors



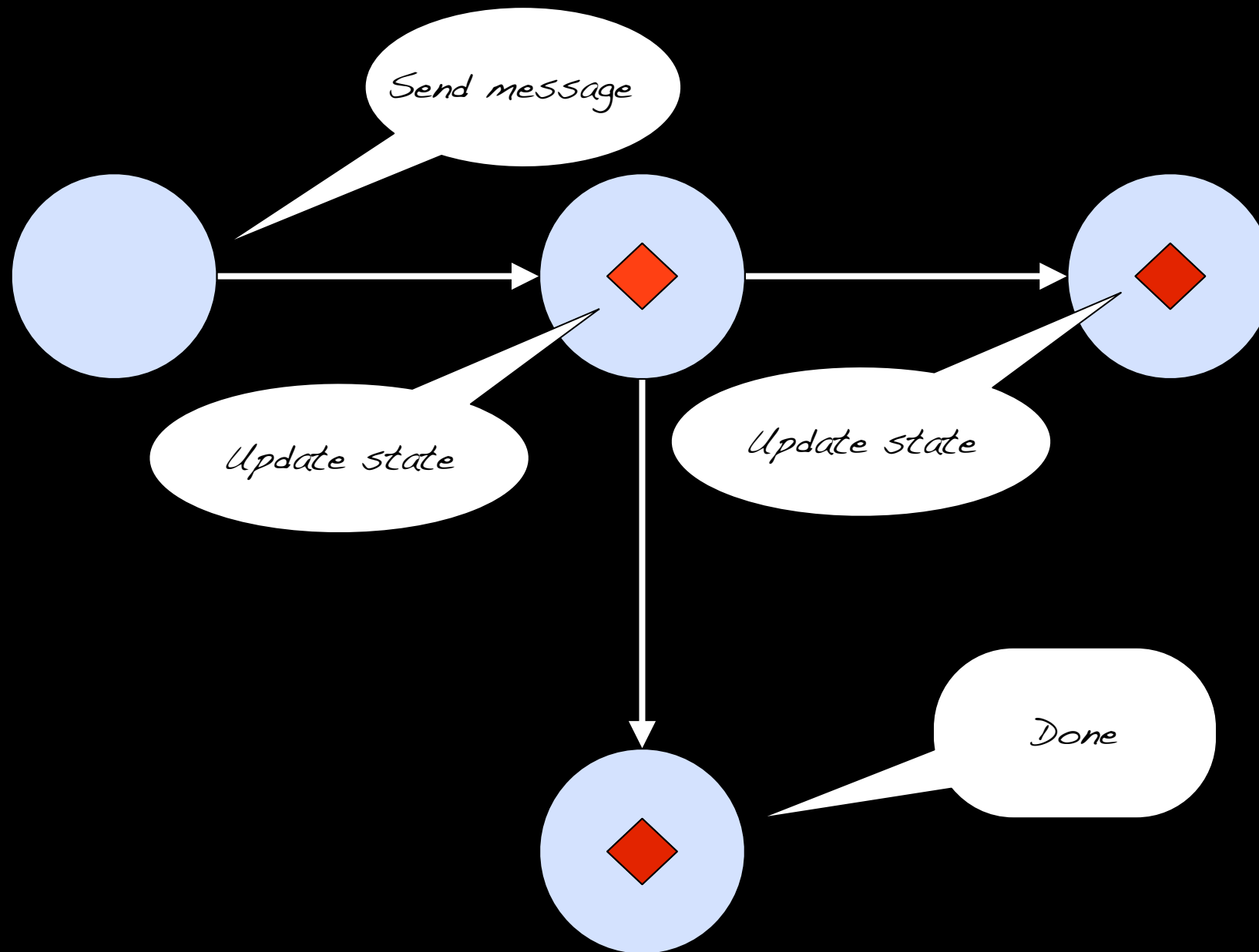
Actors



Actors



Actors



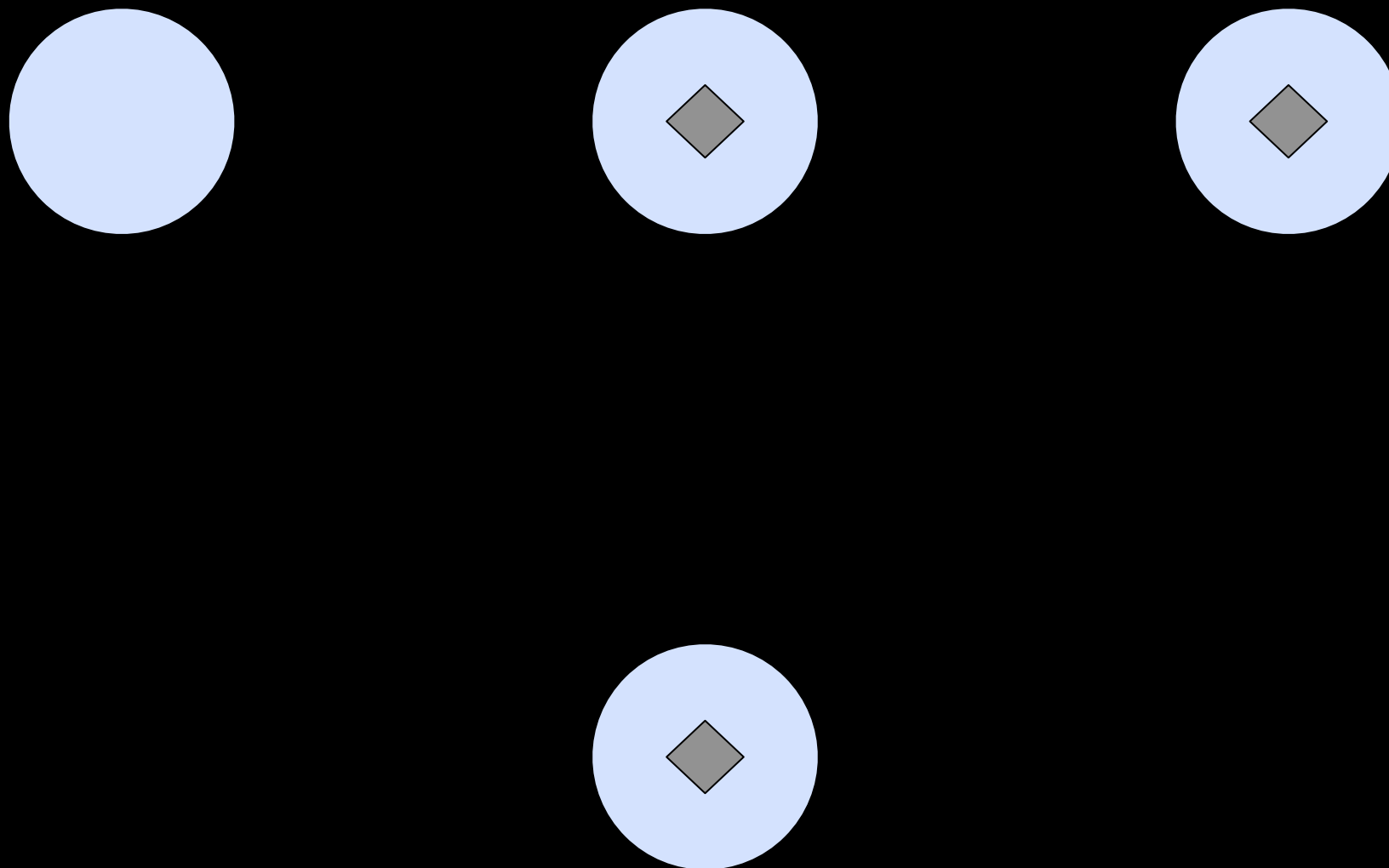
REPL time!

Actor gotchas

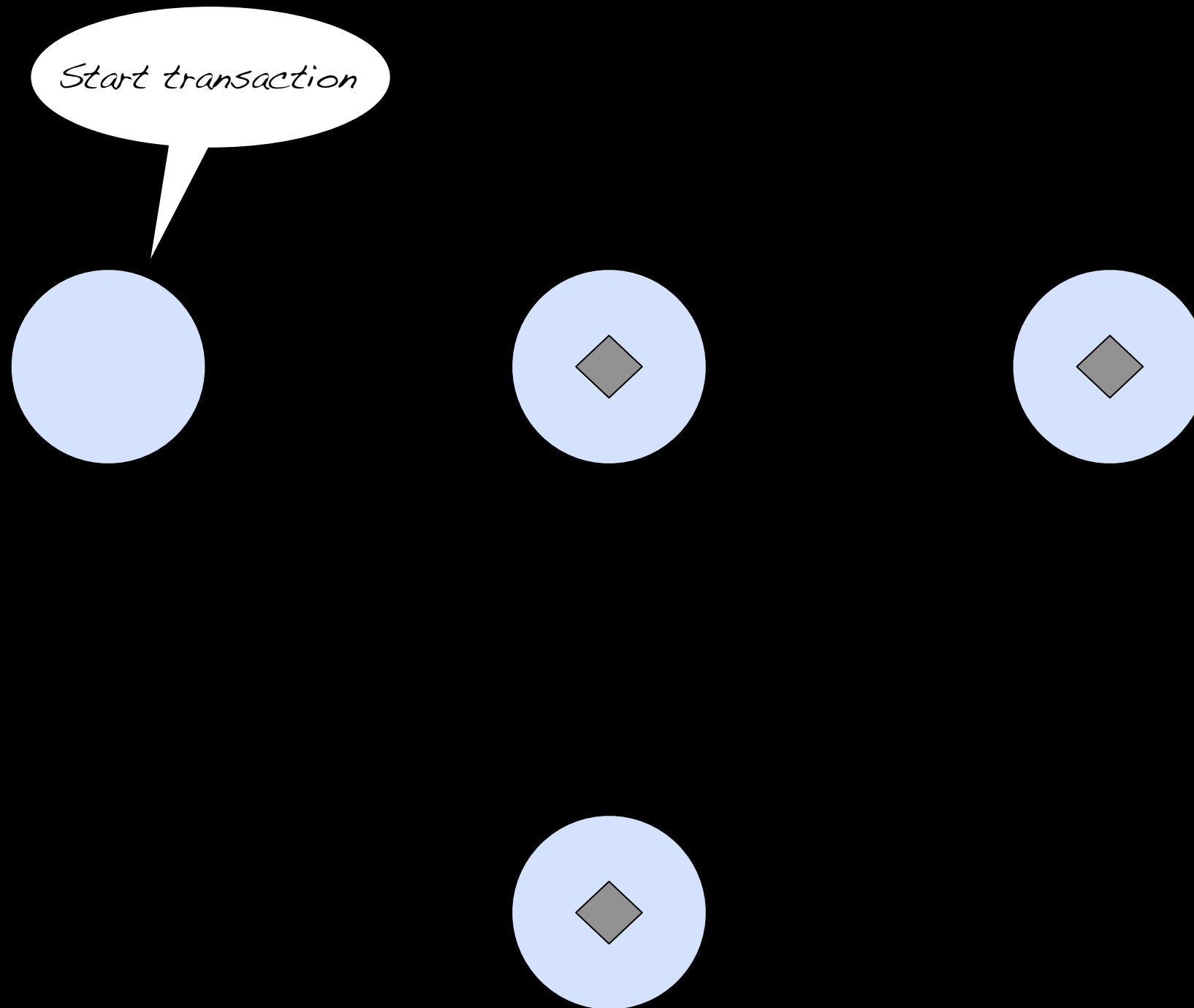
- Messages should be **immutable**
- References to mutable internal state must not escape the Actor
- Coordinating state changes between multiple actors is **hard**
- **Actors are reactive**, not proactive

Actors + STM =
Transactors

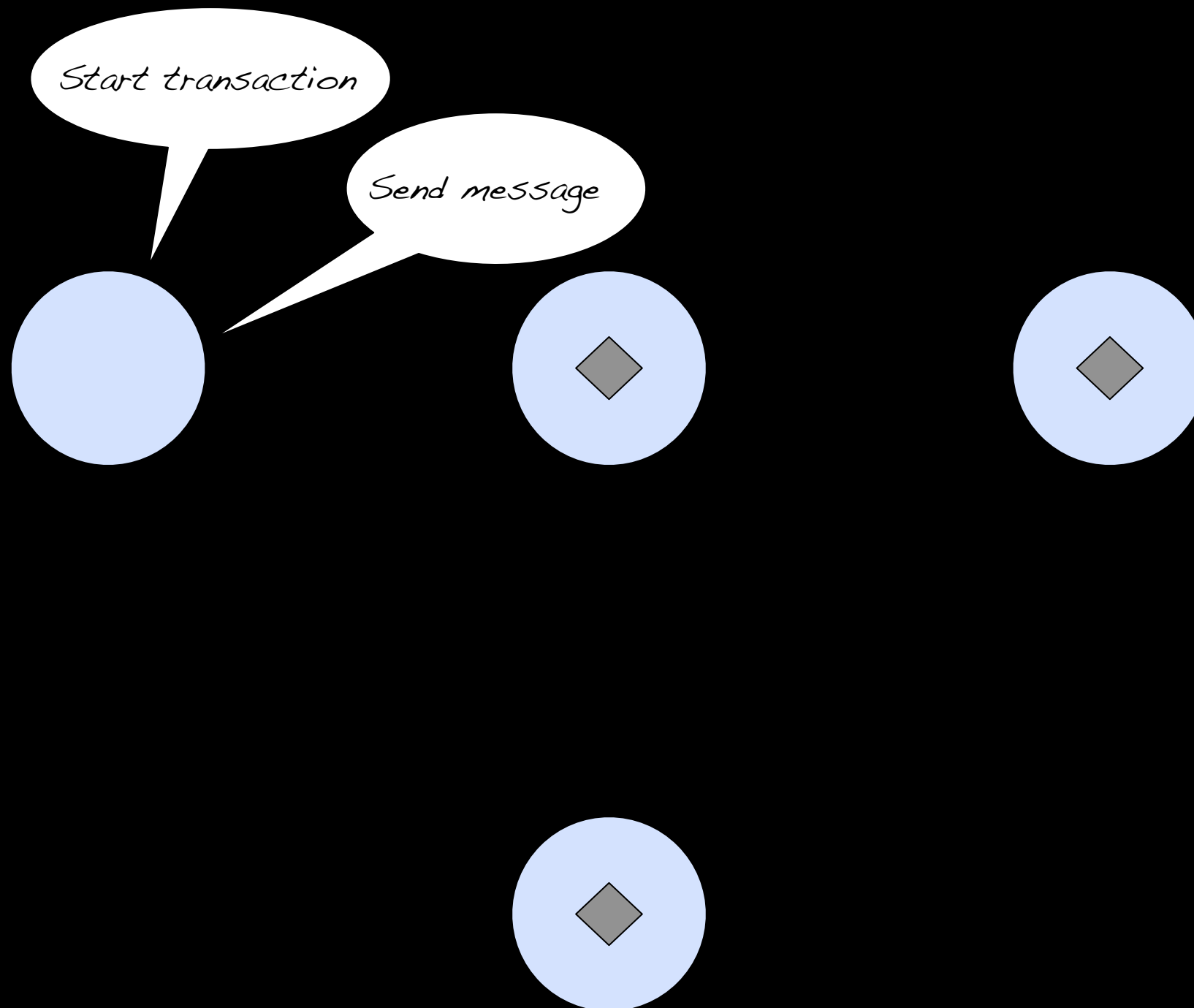
Transactors



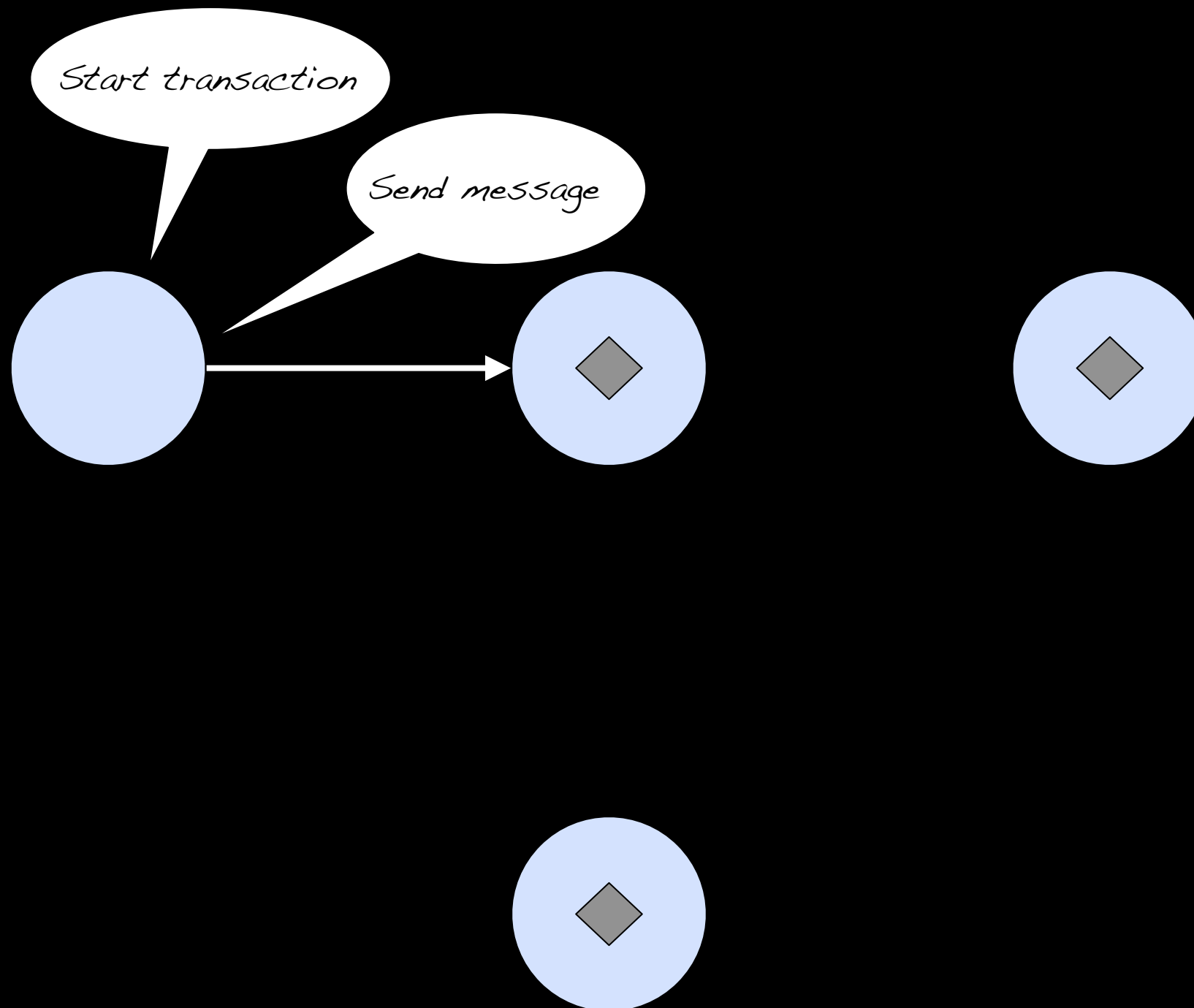
Transactors



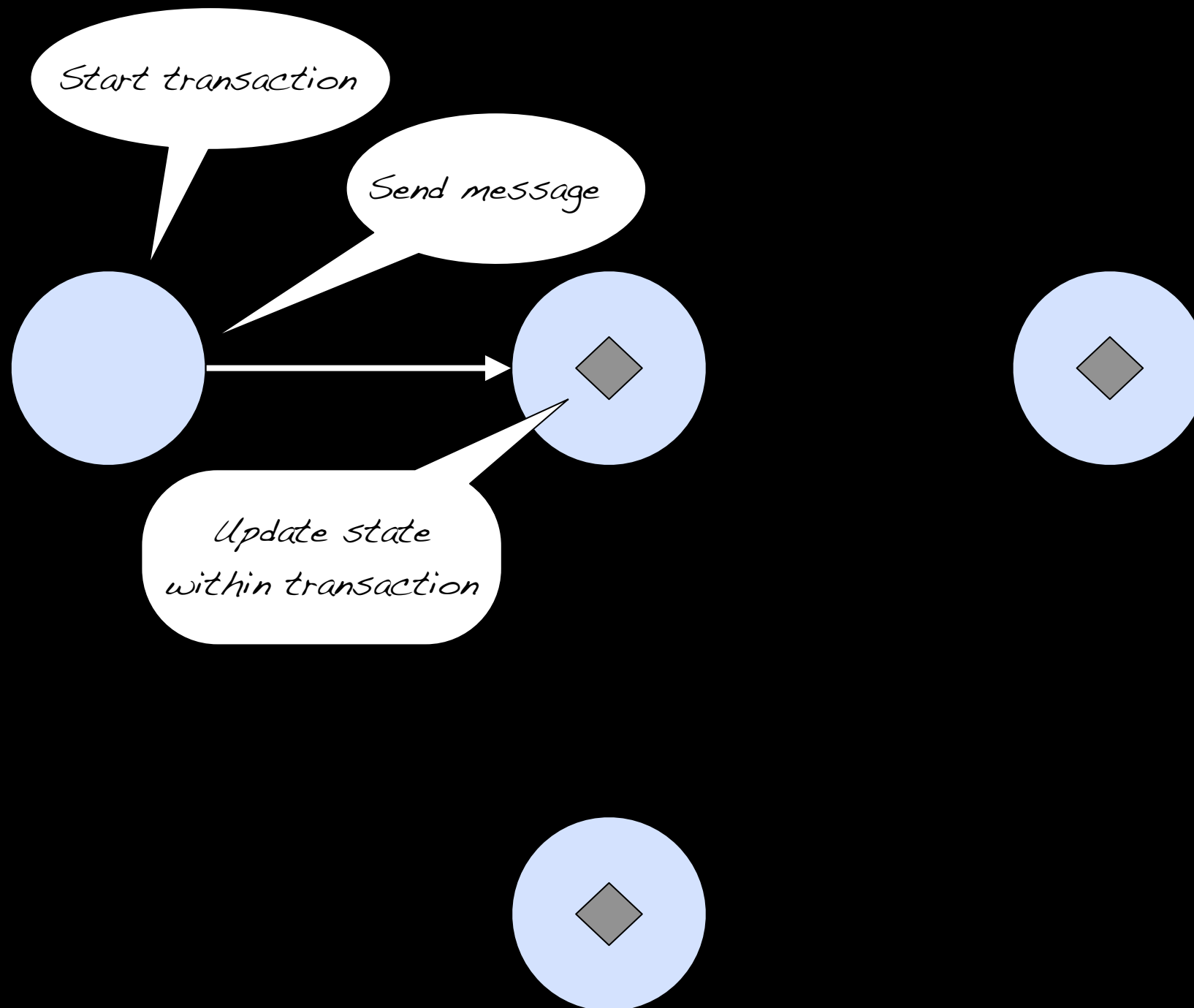
Transactors



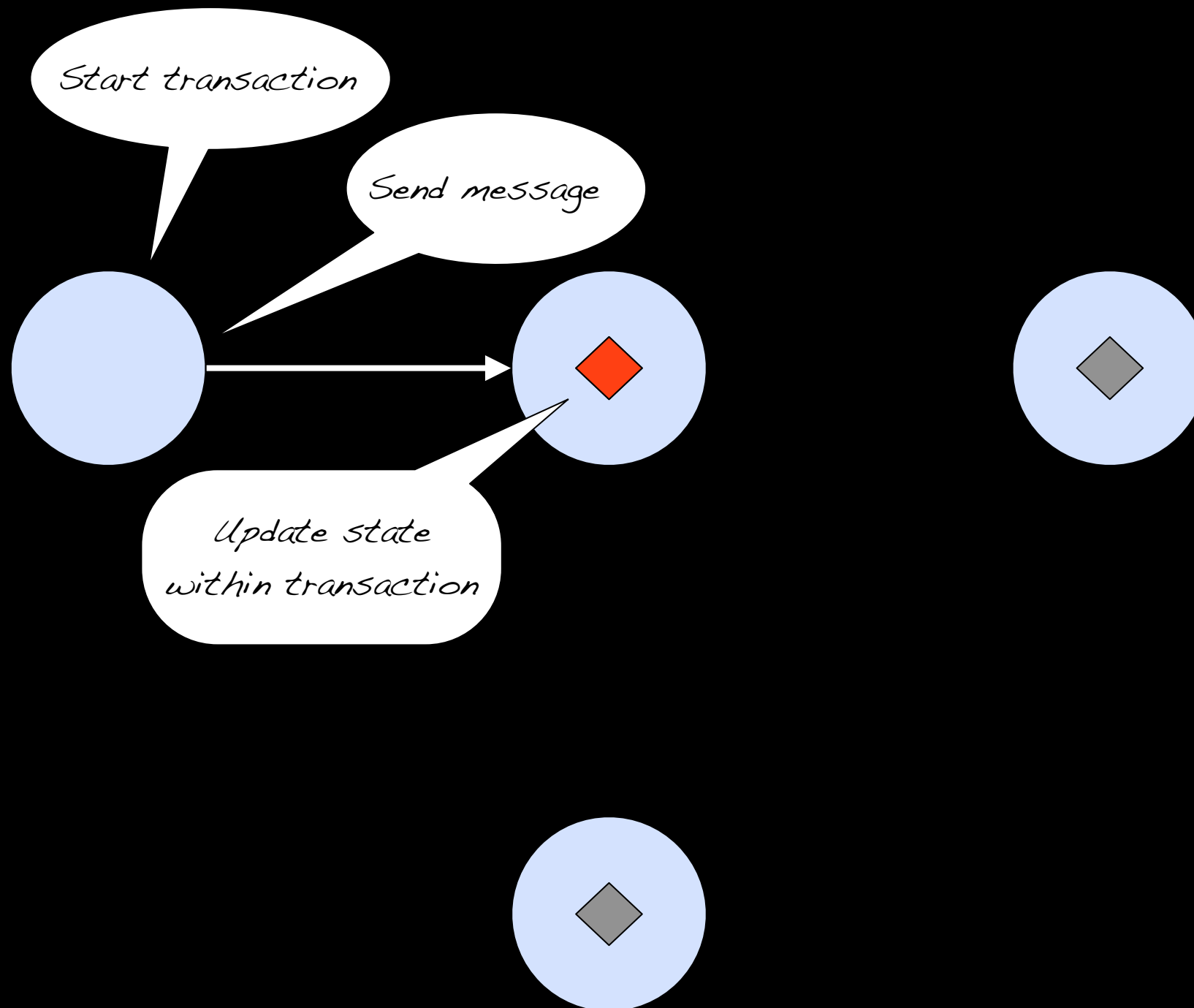
Transactors



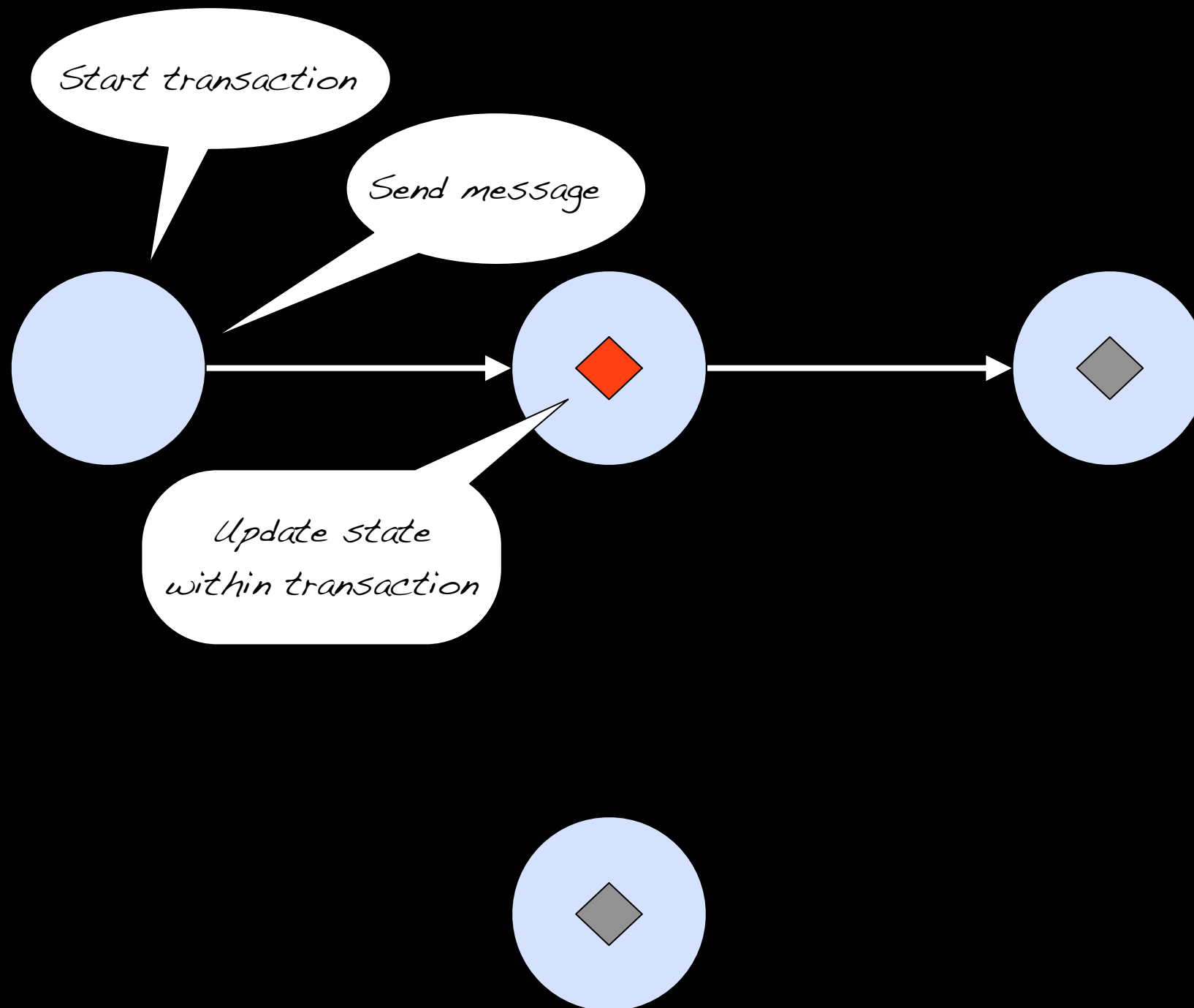
Transactors



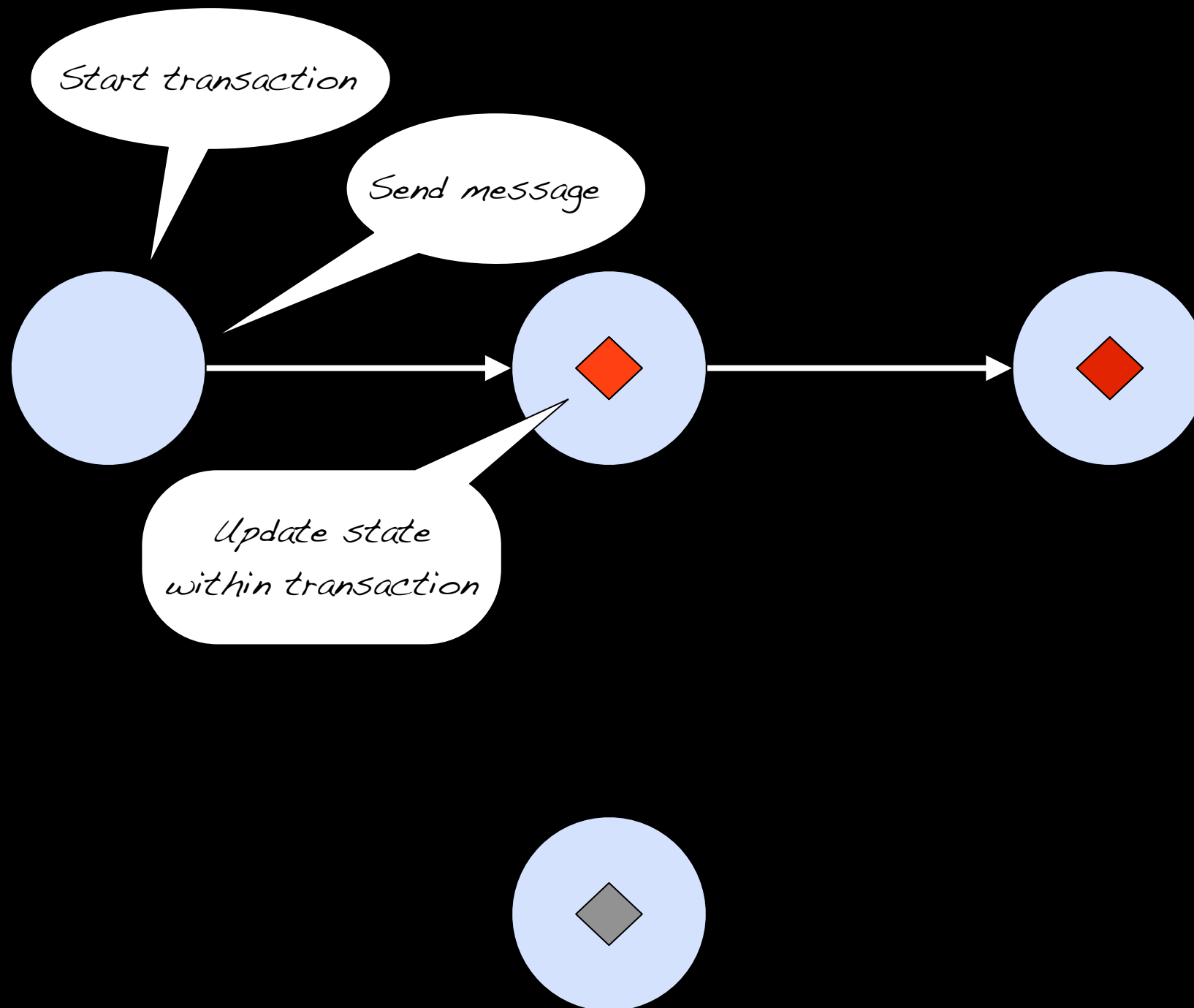
Transactors



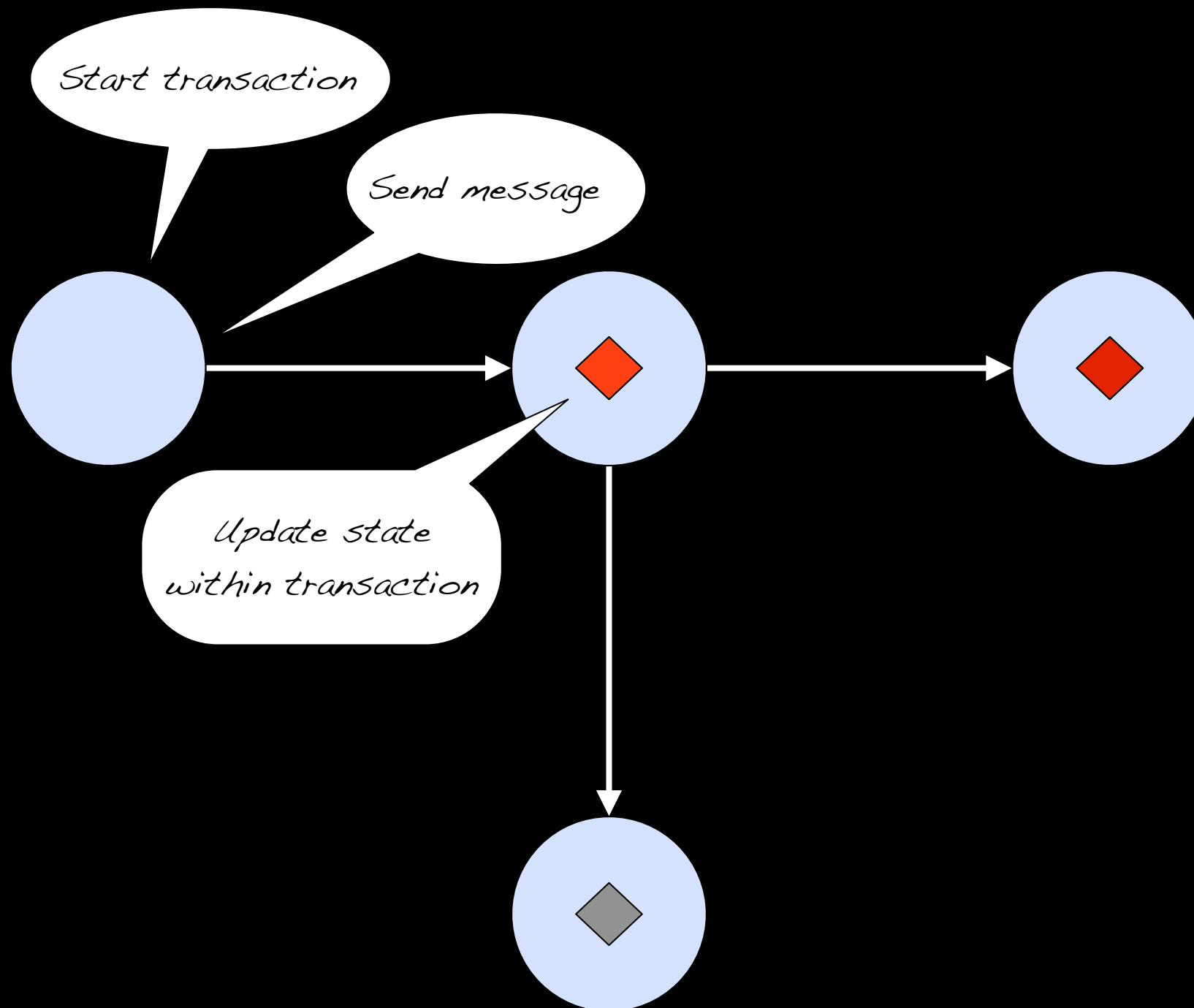
Transactors



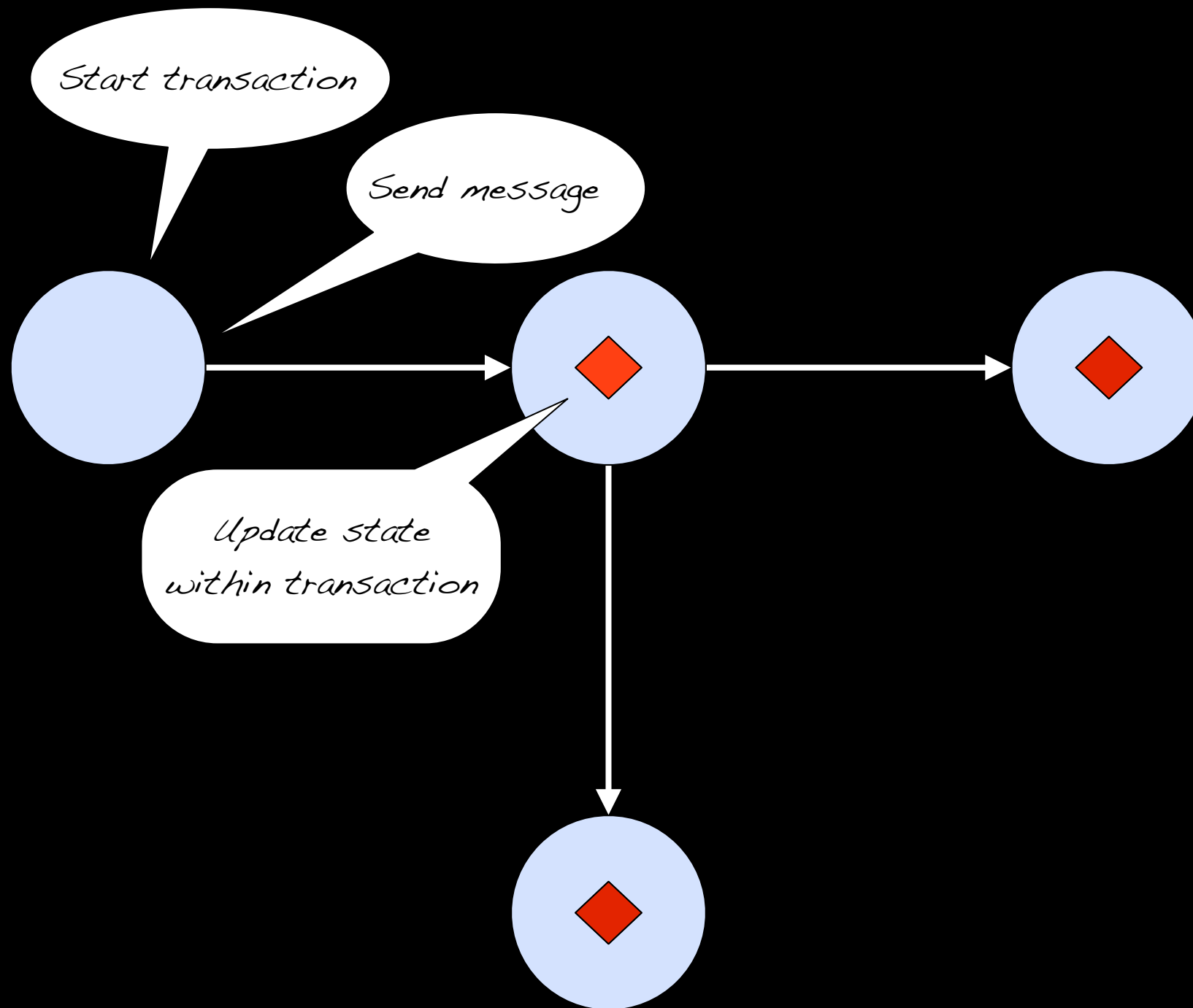
Transactors



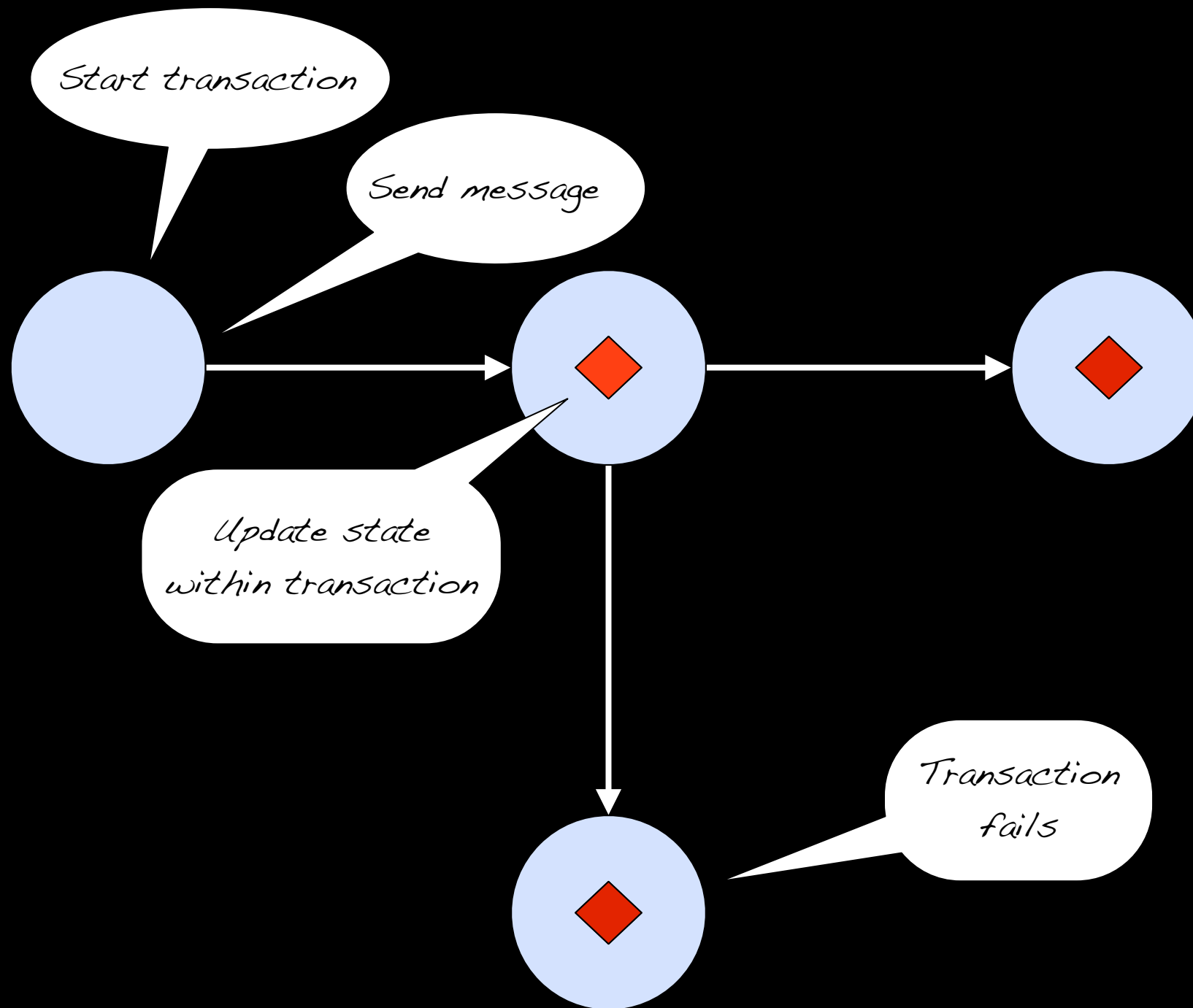
Transactors



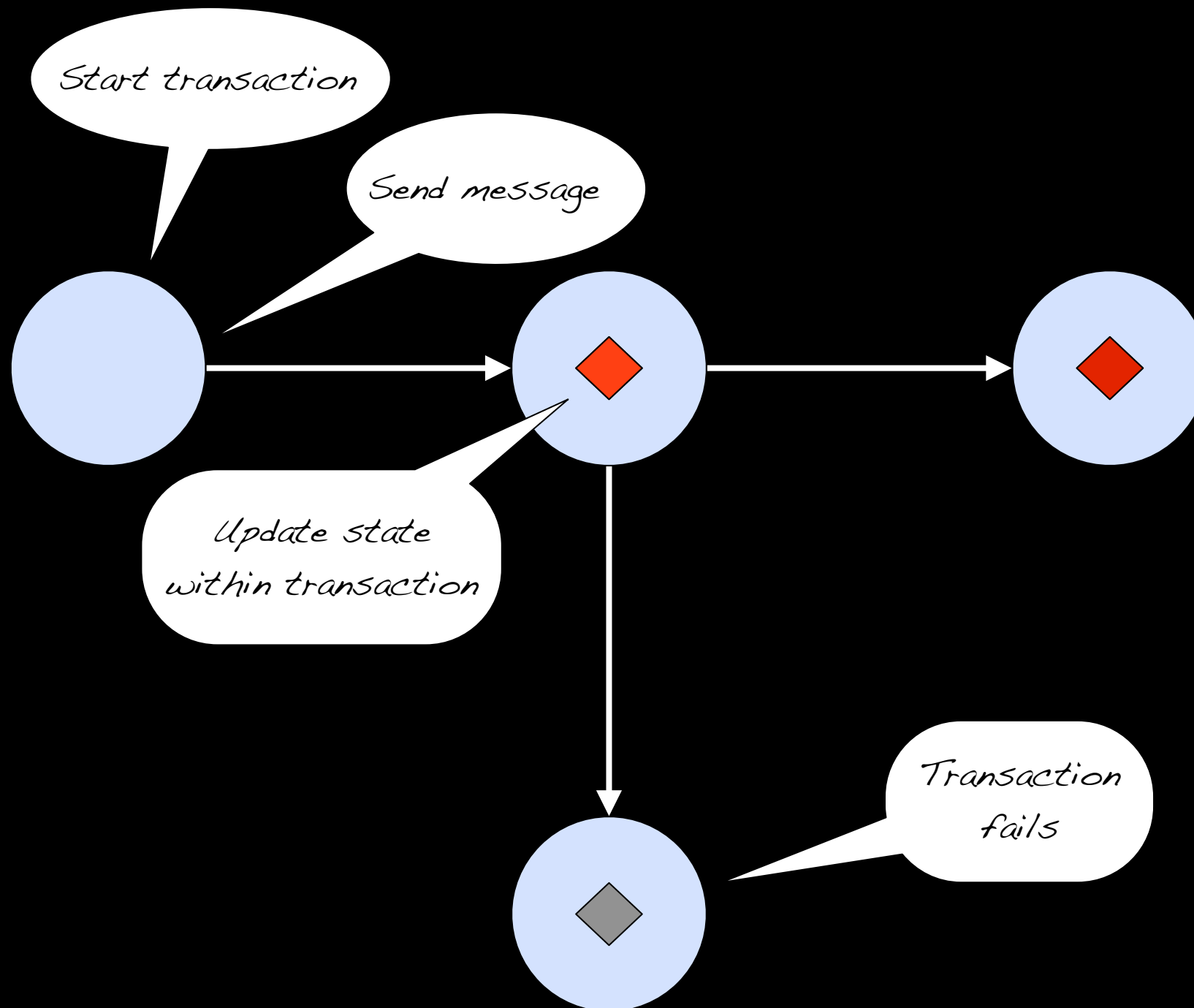
Transactors



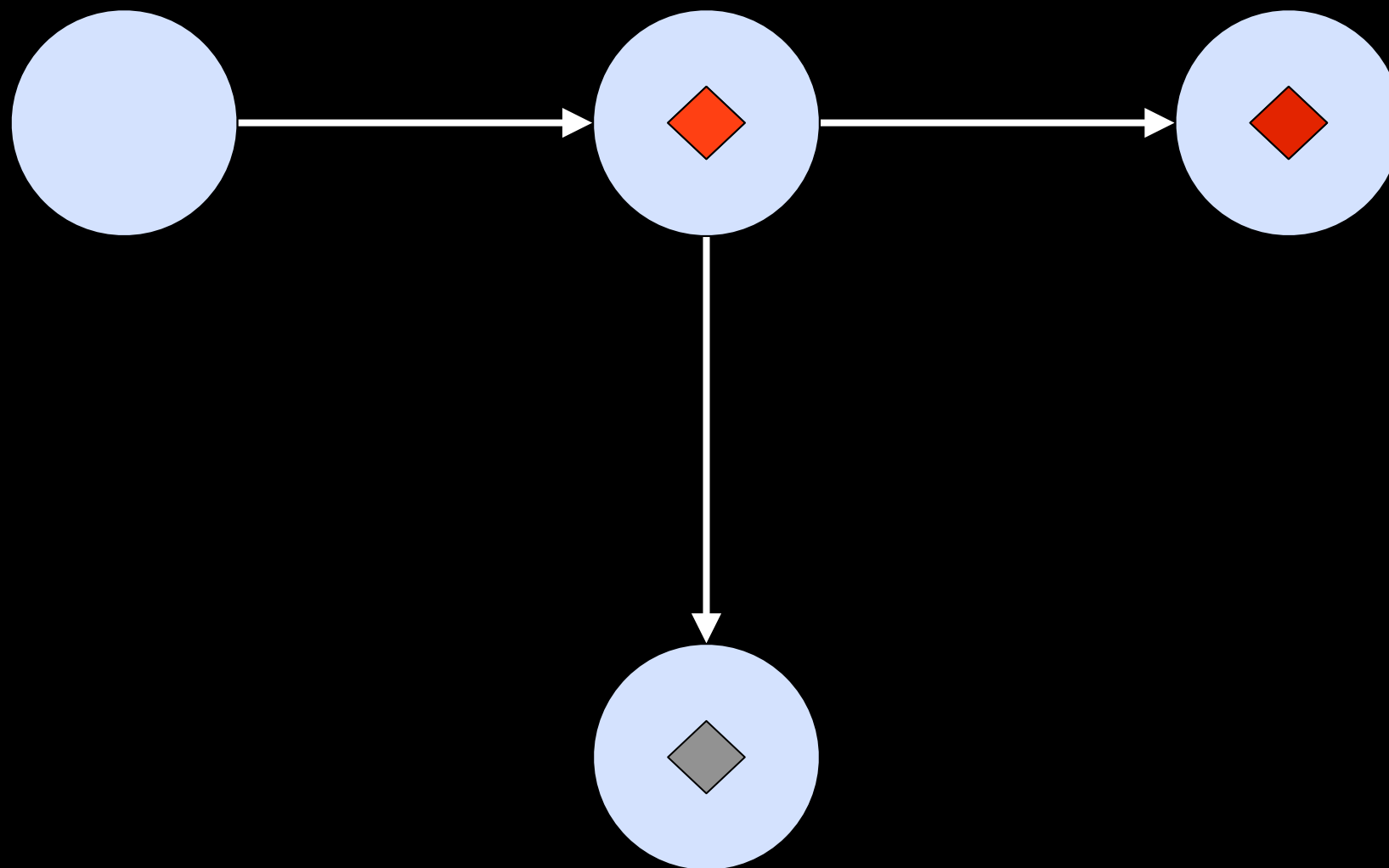
Transactors



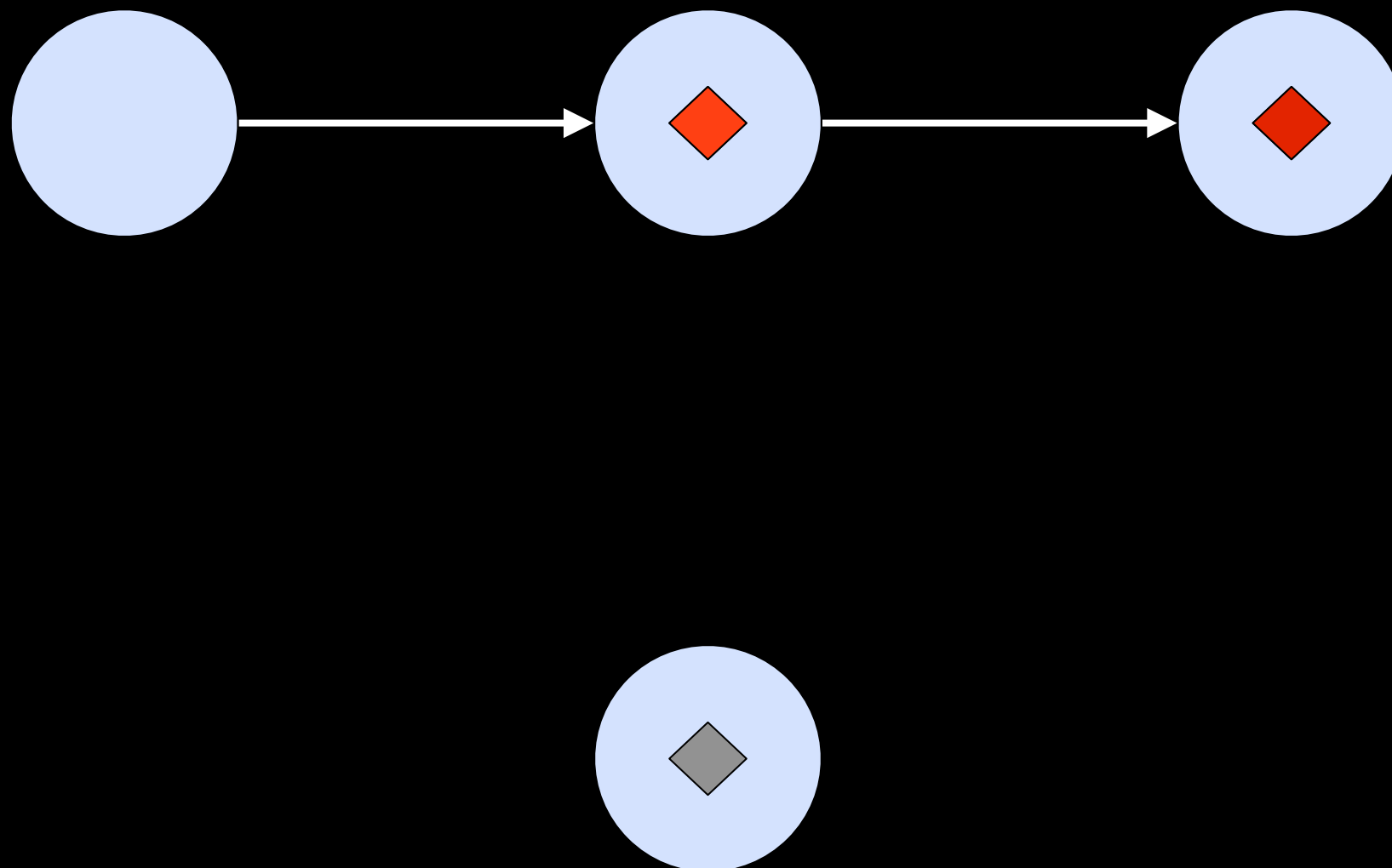
Transactors



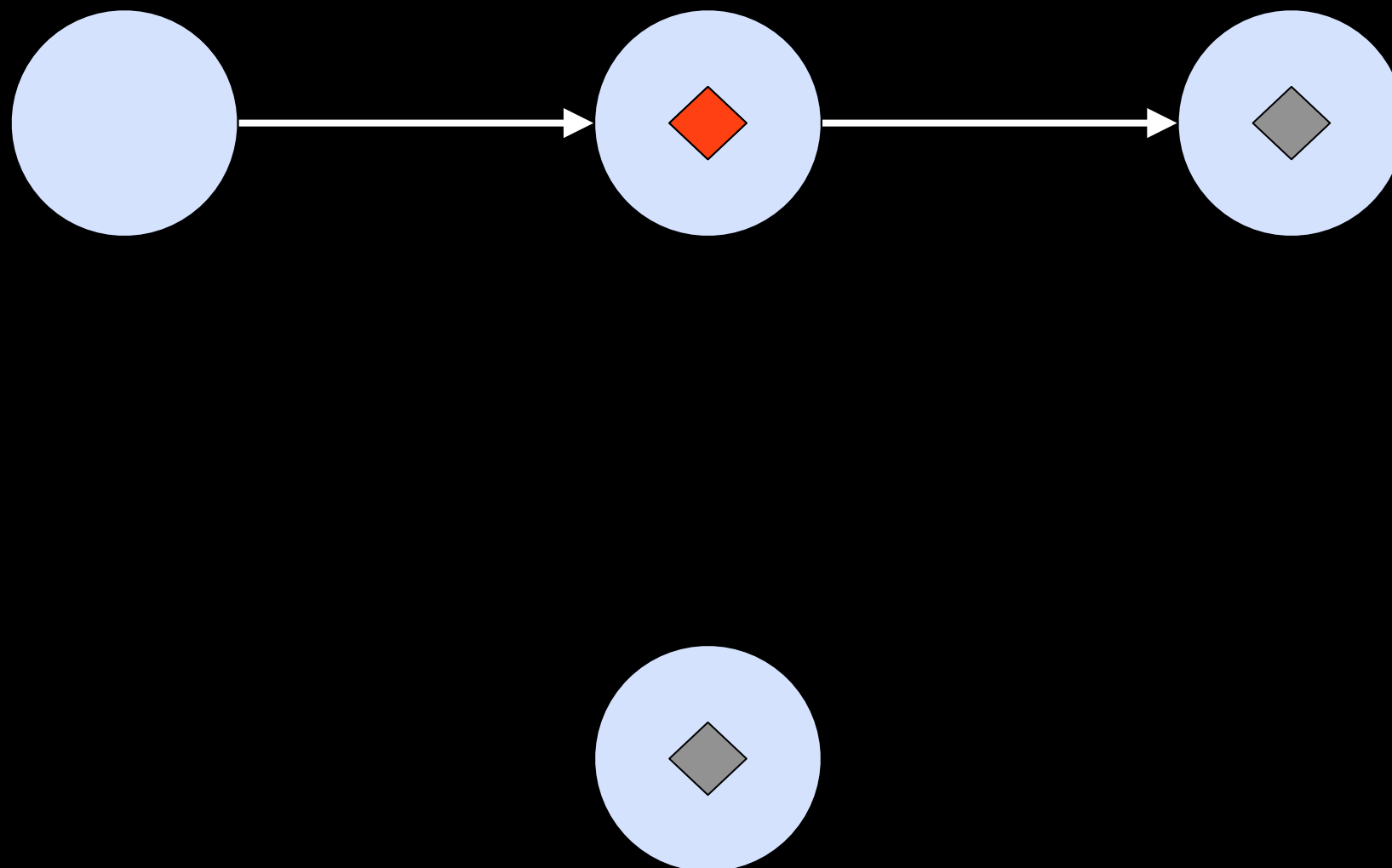
Transactors



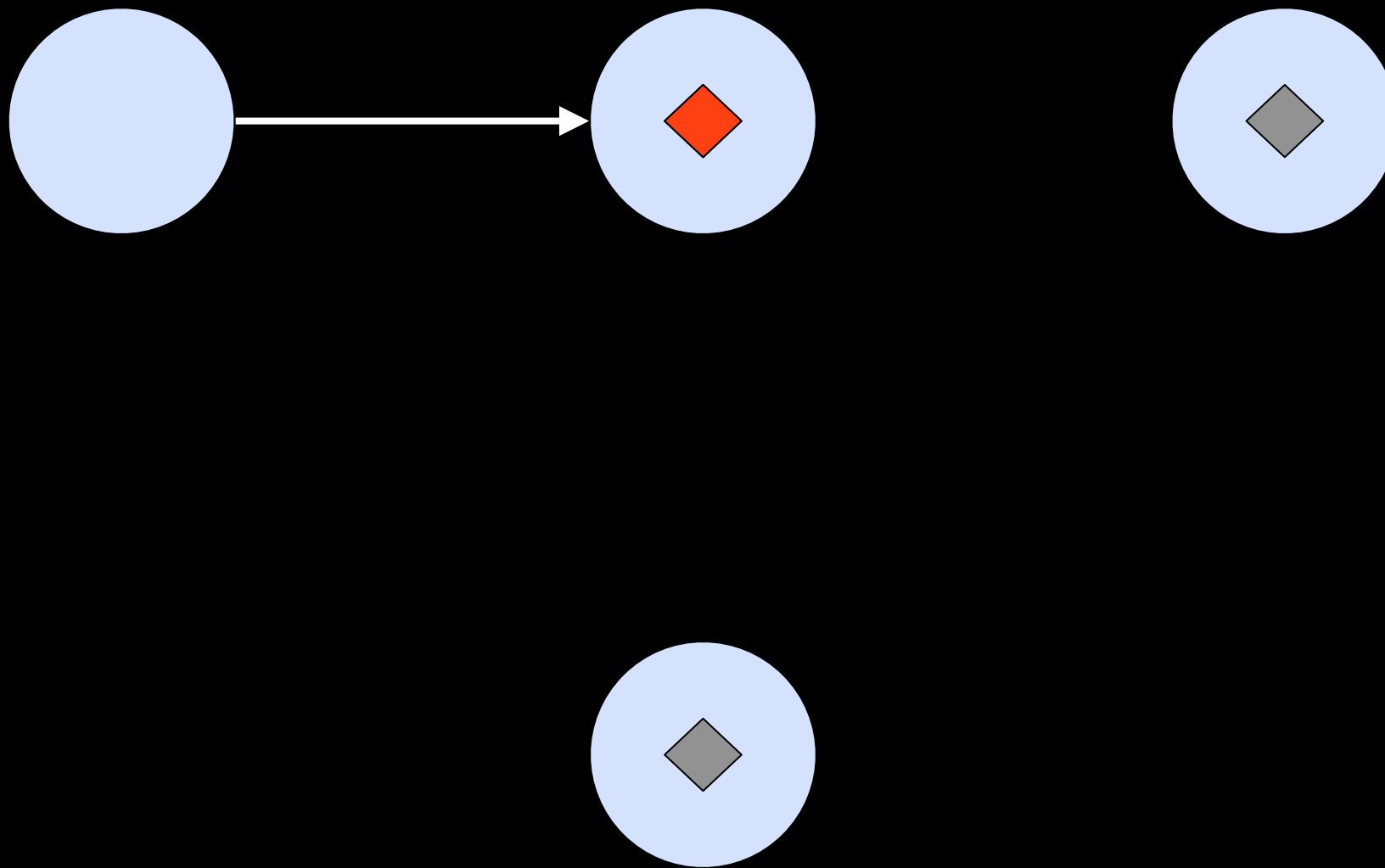
Transactors



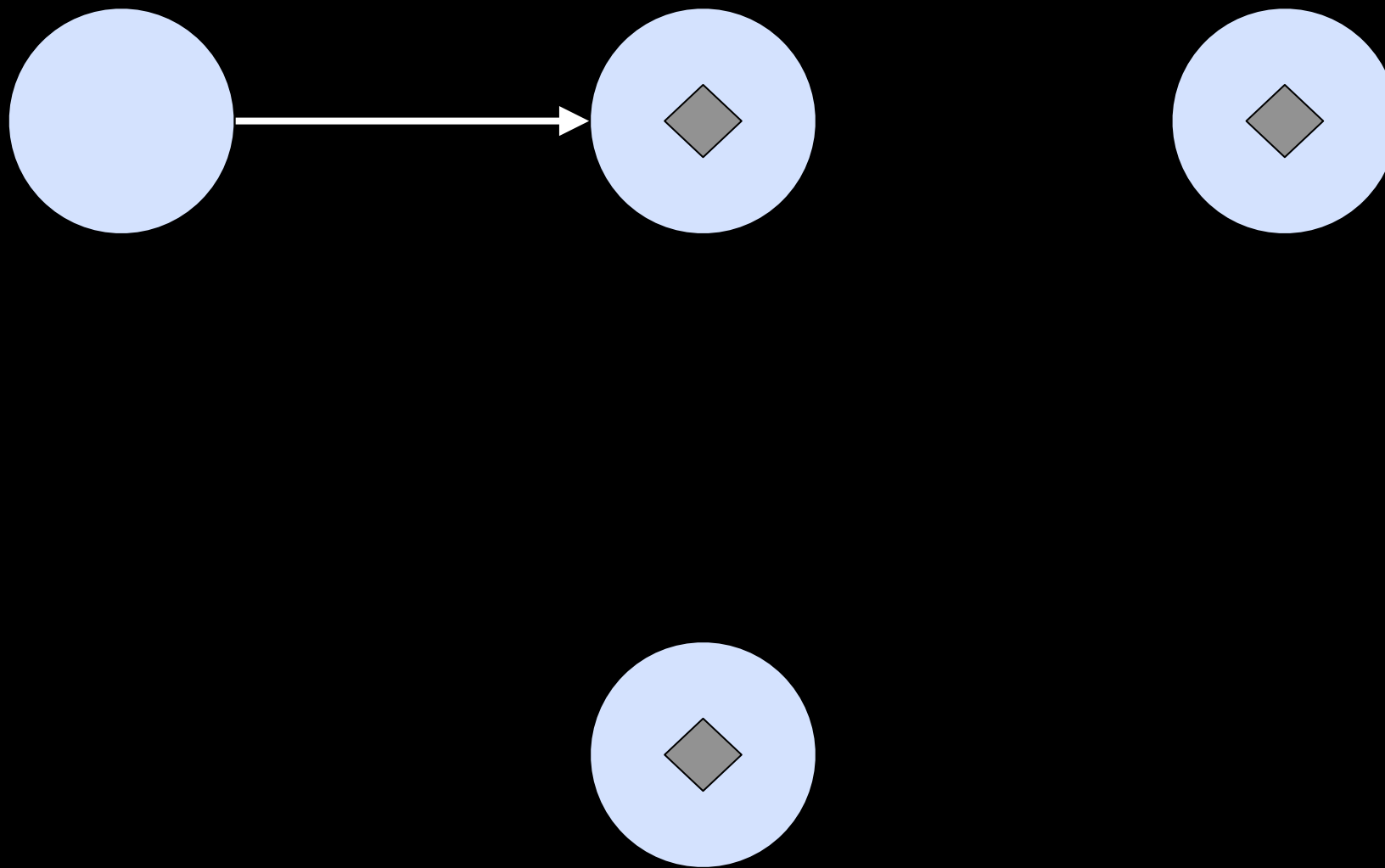
Transactors



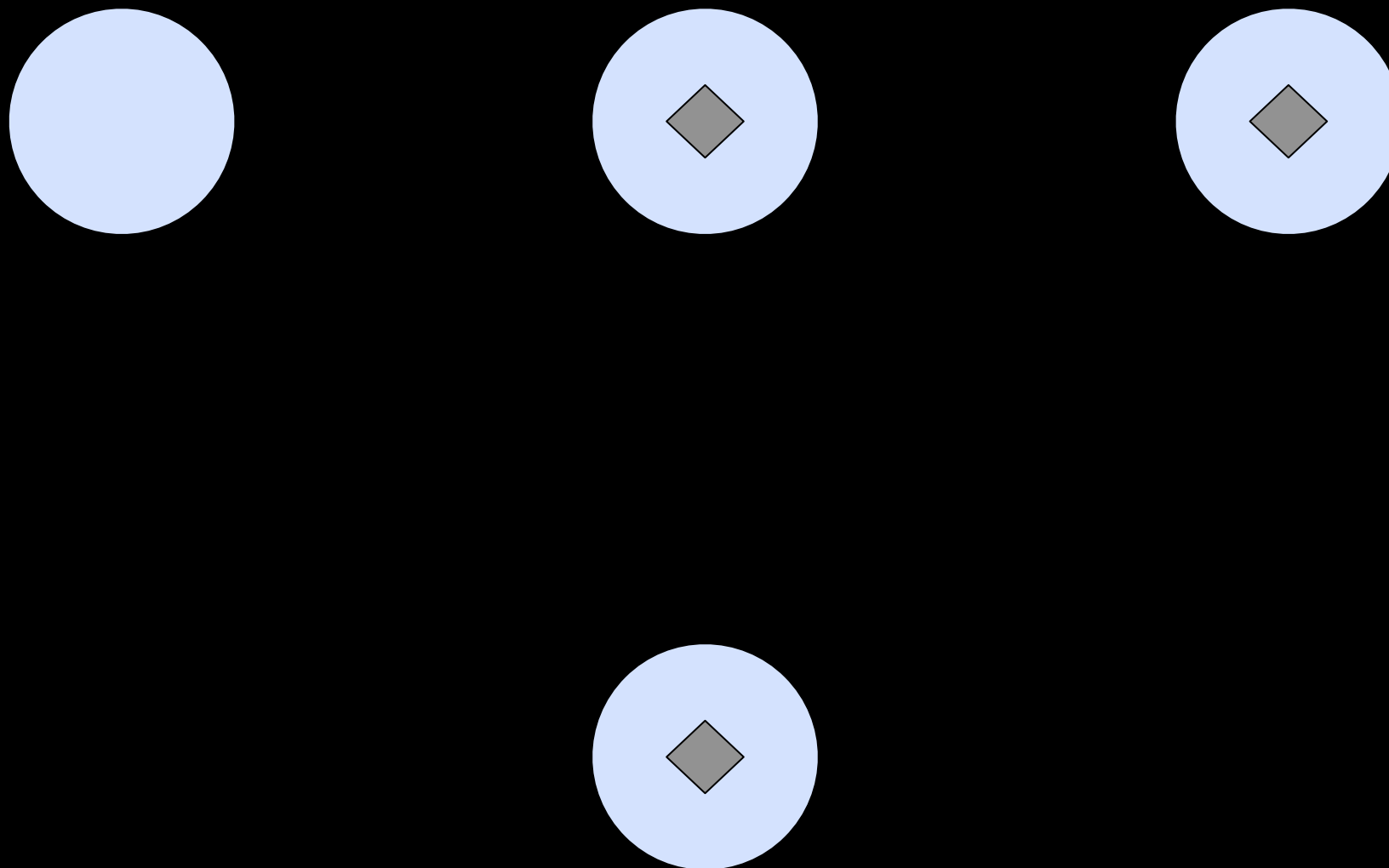
Transactors



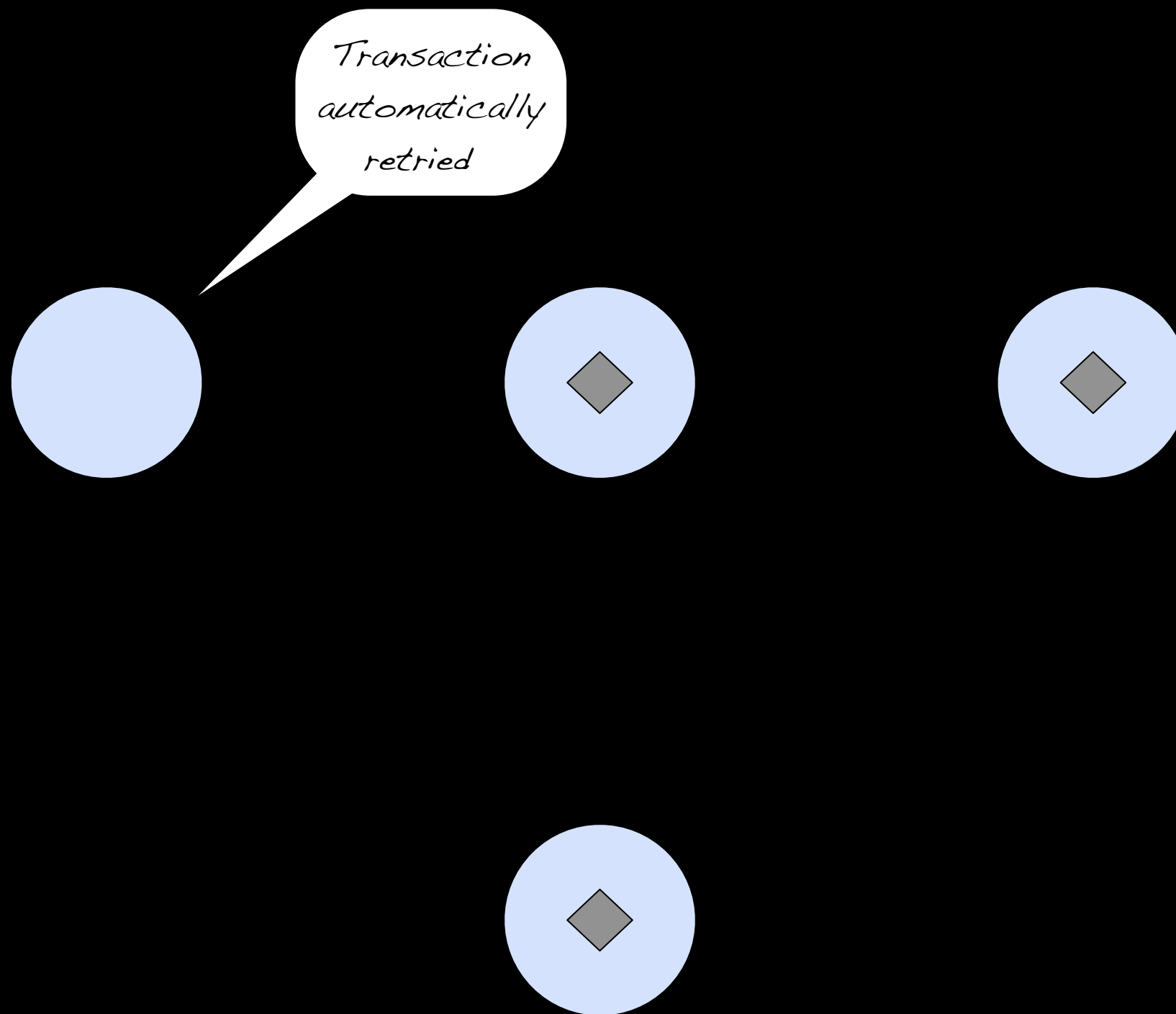
Transactors



Transactors



Transactors



REPL time!

Transactor Overview

- All the good from **Actors**
- All the good from **STM**
- **Removes** state update **coordination issues** from Actors
- **Replaces Threads** as the concurrency mechanism for STM

Transactor gotchas

- Non-transactional changes may occur if such code is called within a transaction
-

Summary

Threads and locks
are...

...sometimes

plain evil

It doesn't have to
be like this

We need to raise the
abstraction level

...but **always** the

wrong default

Introducing



STM

Actors

Agents

Dataflow

Distributed

Open Source

RESTful

Secure

Persistent



www.akkasource.com

The Directors Cut

*The Dining
Hackers*