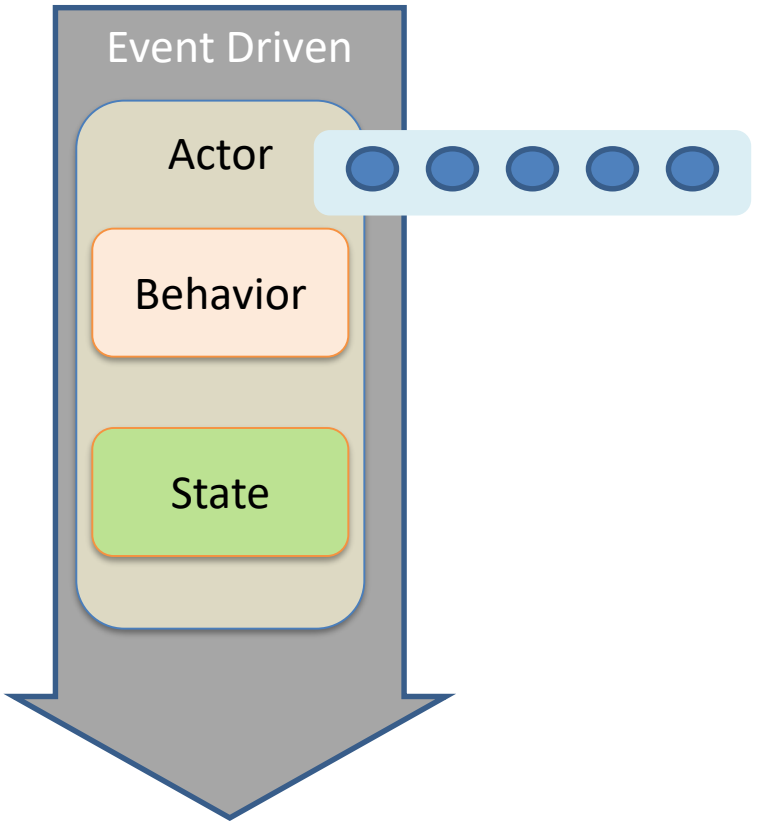
Akka Essential

Akka

- AKKA is a toolkit and runtime for building highly concurrent distributed and fault tolerant event driven application on the JVM
- Parallism
- Concurrency



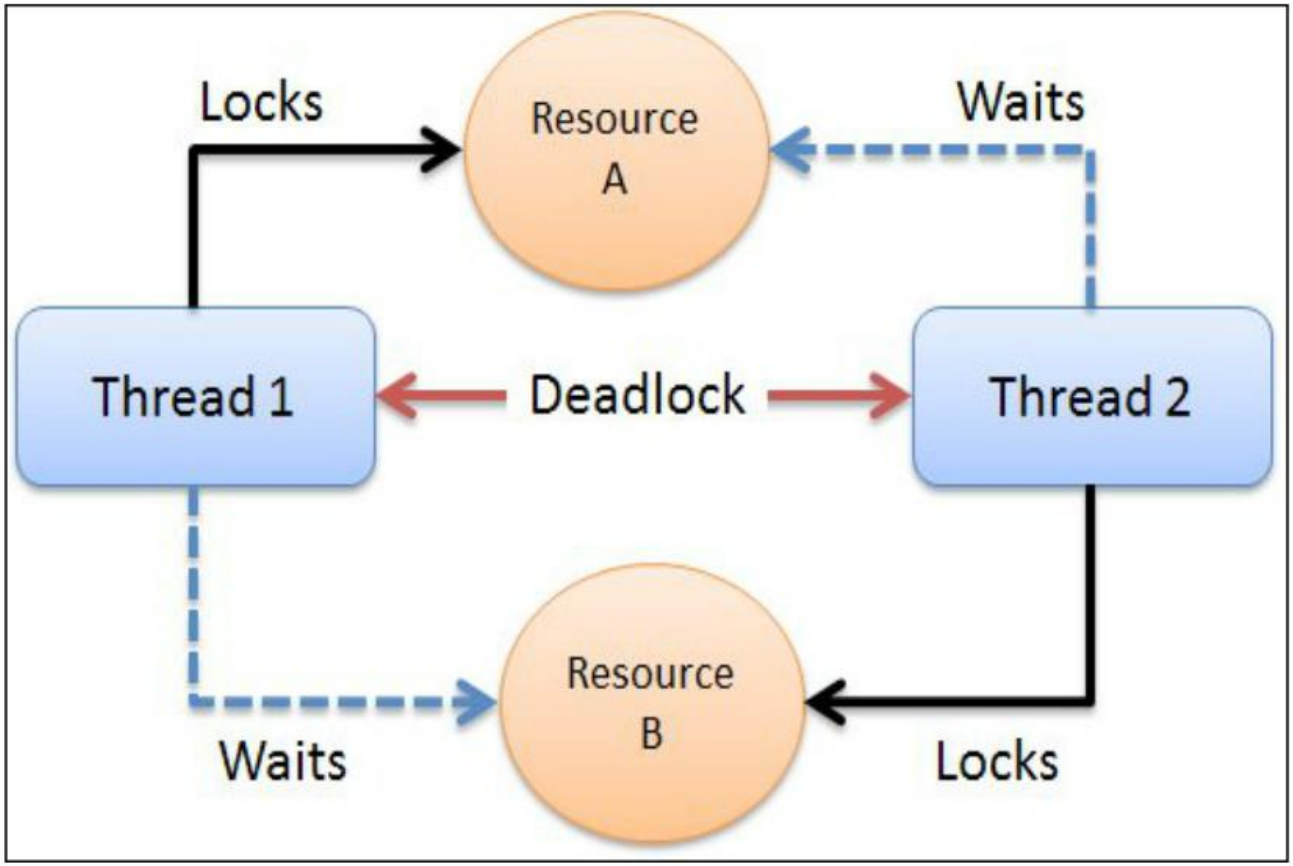
Akka Features

- **Concurrency:** abstract concurrency handling and allows to focus on the business logic
- **Scalability:** asynchronous message passing allows applications to scale up on multicore servers
- **Fault tolerance:** use from Elrang „Let It Crash” model
- **Event-driven architecture:** provides asynchronous messaging patform for building event-driven architecture

Akka Features

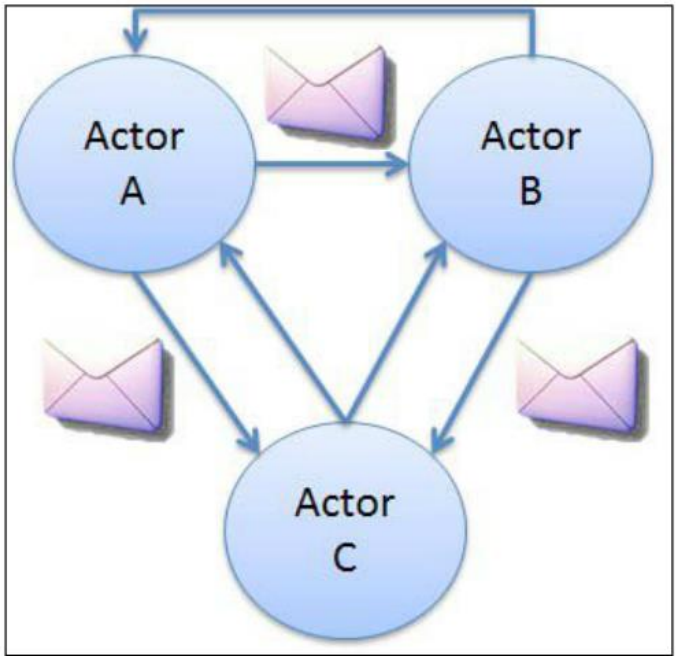
- **Transaction support:** implements transactors that combine Software Transactional Memory (STM) into transactional actors
- **Location transparency:** uniform programming model for multicore and distributed computing needs
- **Scala/Java API**

Concurrent systems



Working with thread requires much higher level of programming

Actors



There is no problem with threads and locks

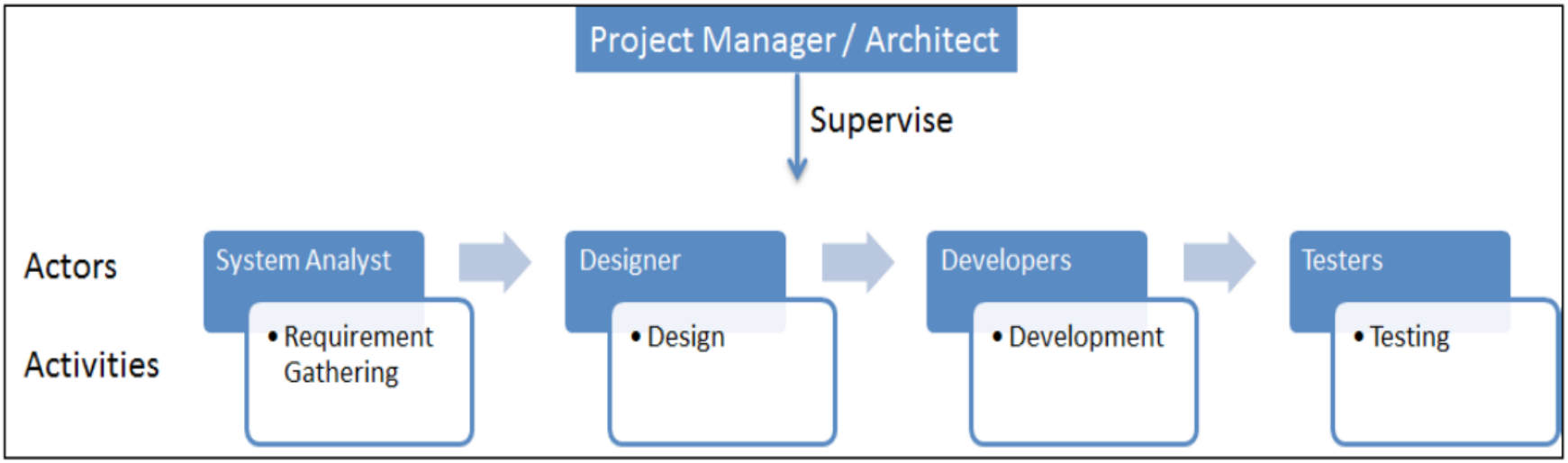
There is no shared state between actors

In 1973, Carl Hewitt, Peter Bishop, Richard Steger wrote paper.
Actor Model was implemented in Erlang by Joe Armstrong in Ericsson

Actor features

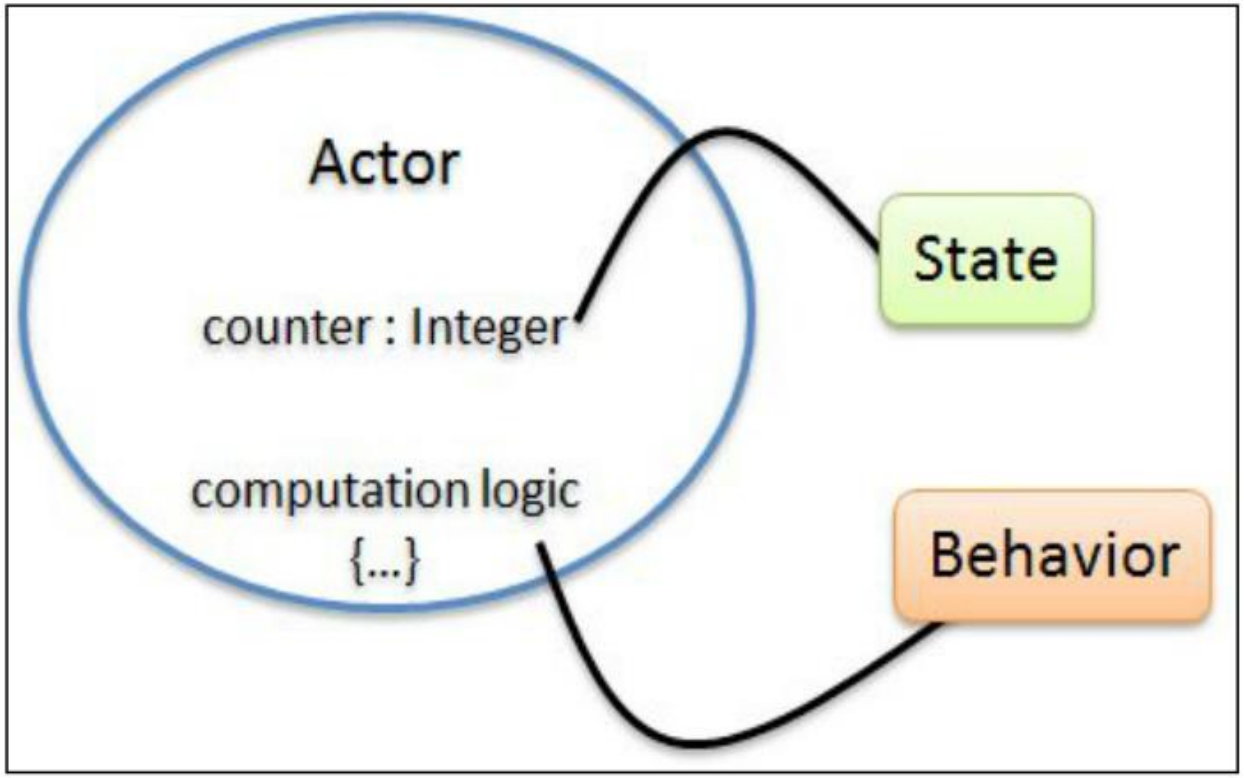
- The immutable messages are used to communicate between actors.
- Actors do not share state.
- Actors control the access to the state.
- Each actor has a queue attached. One message is process at the time.
- An Actor can response to the message by: sending messages, creating new set of actors, updating its own state, changing computational logic.
- An Actor can send a message with no guarantee on the sequence and execution
- Communication is asynchronous and sending and receiving is performed in different threads.

Actor Systems



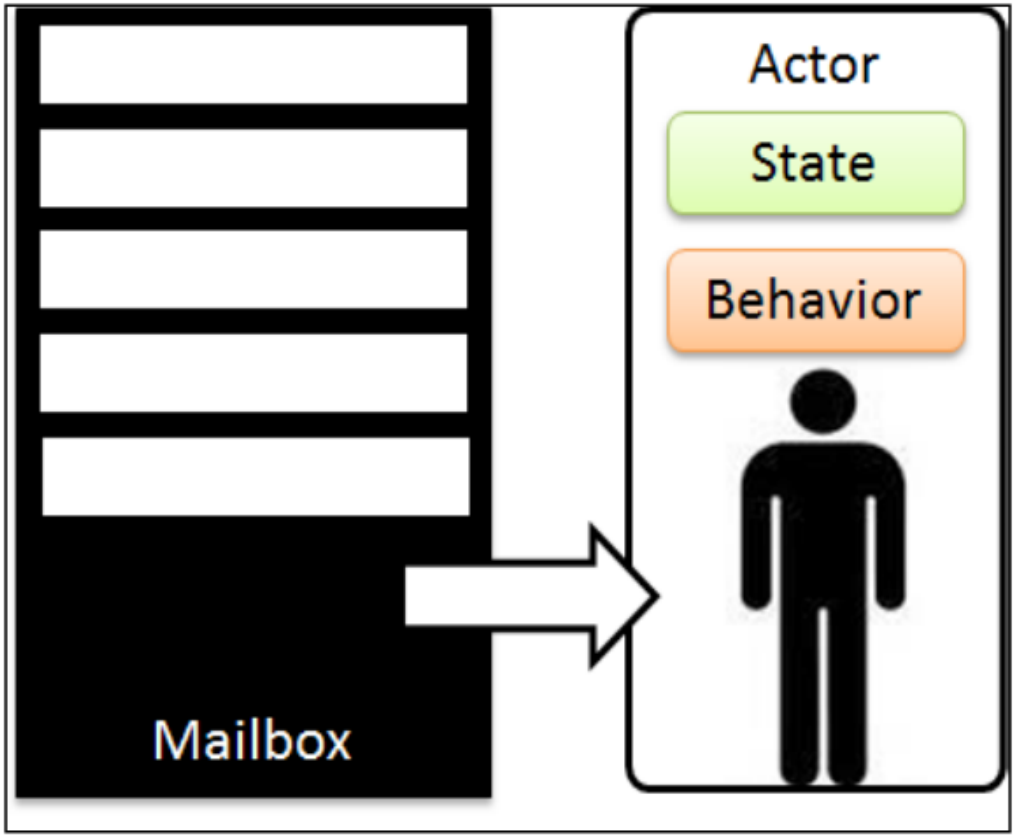
In Akka world, the project is equivalent to the actor systems

What is an Actor ?



The actor behaviour could be changed a new behaviour.
The actors return to the original behaviour after restart.

Mailbox

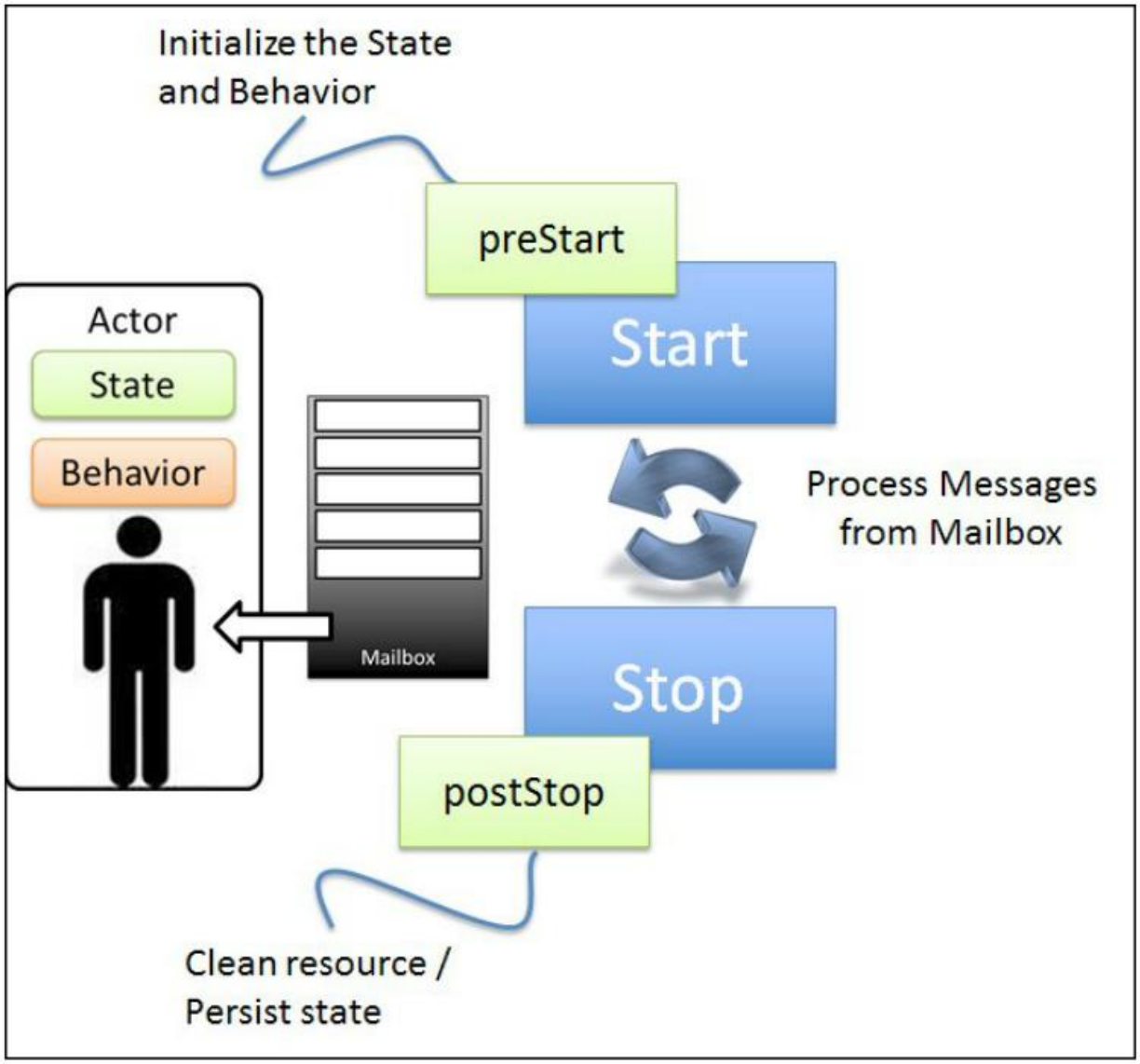


Actor is attached to only one mailbox. The order of arrival of messages are determined by time order of the send operation.

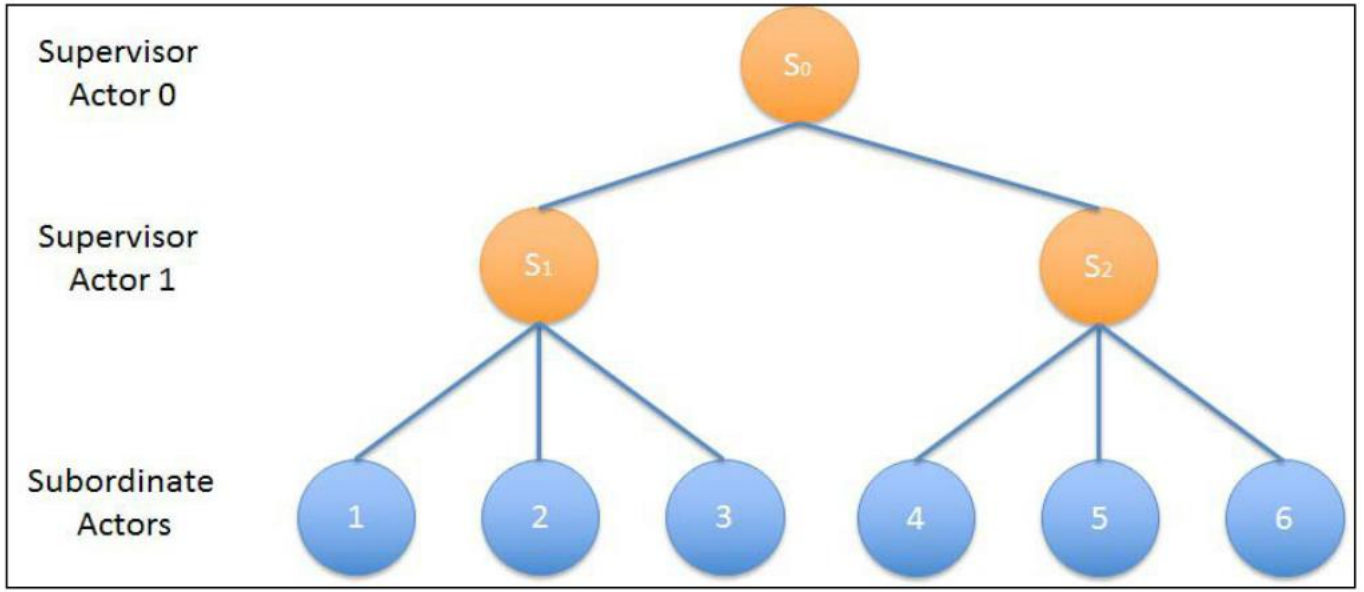
Mailbox

- A bounded/unbounded
- Priority mailbox

Actor lifecycle

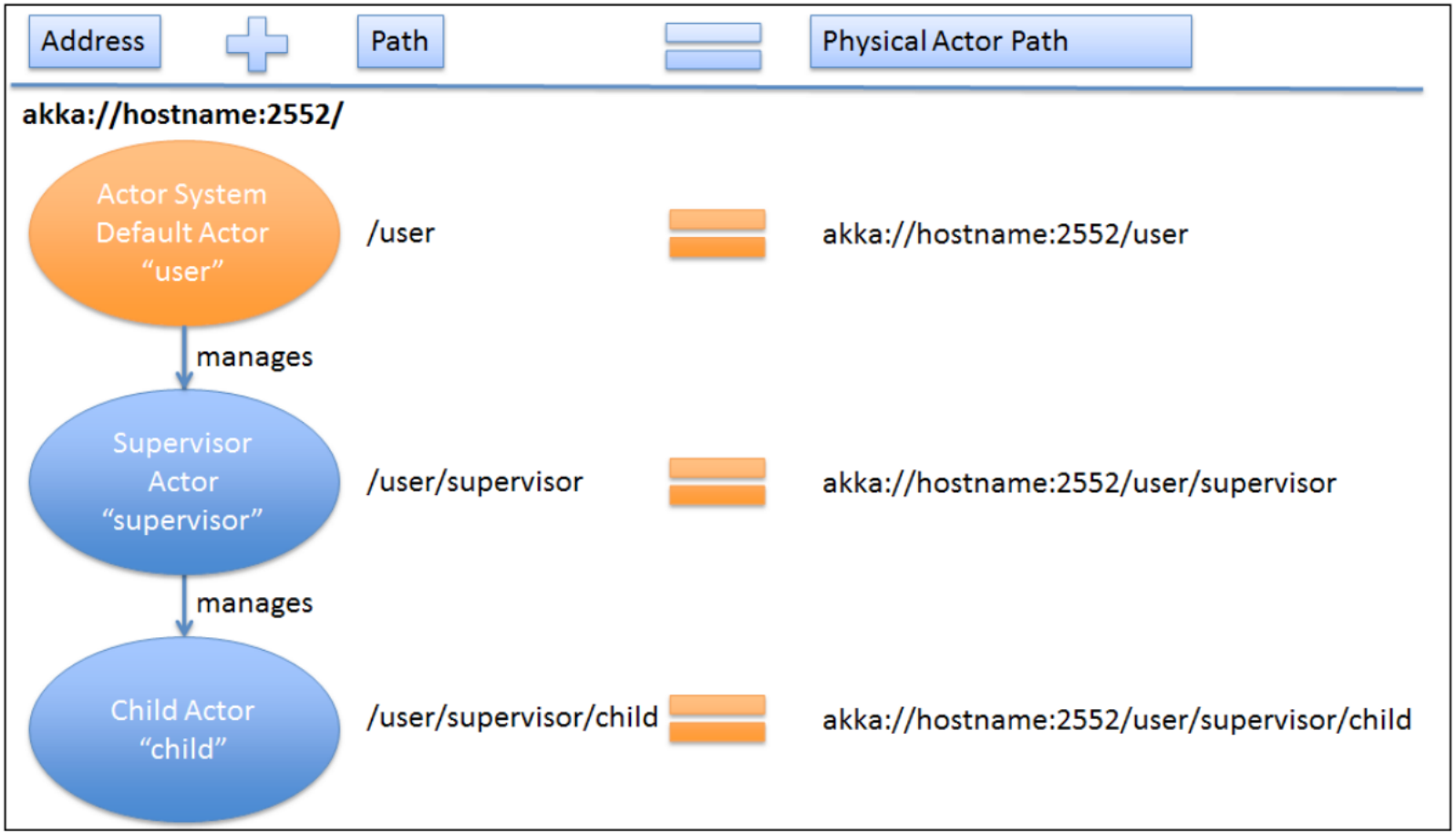


Faul tolerance

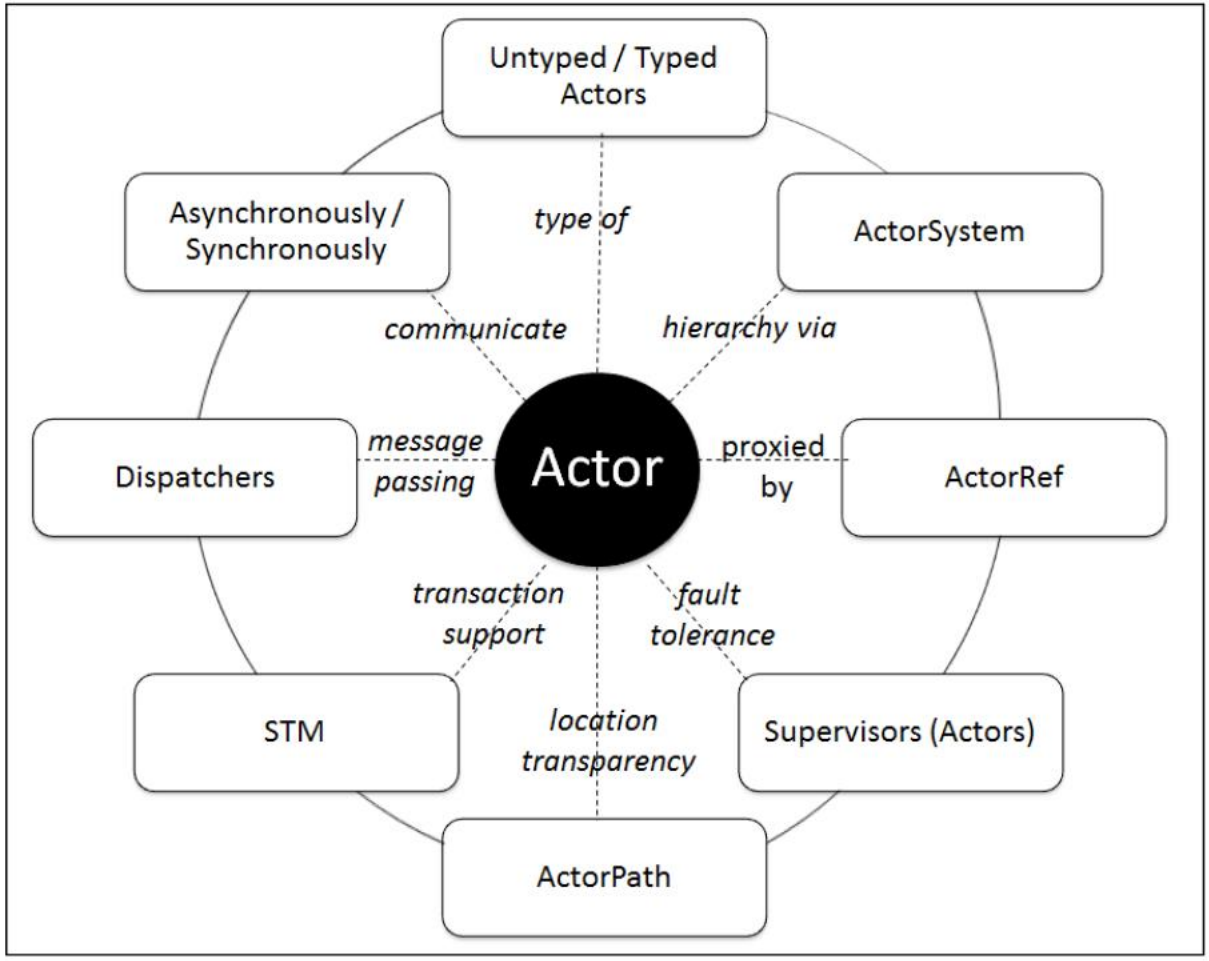


If the actor does not know how to handle a particular message, the actor asks its supervisor for help

Location Transparency



Basis of Akka



Actors Features Summary

- An **actor** is a computation unit with state, behavior, and its own mailbox
- There are two types of actors—untyped and typed
- Communication between actors can be asynchronous or synchronous
- Message passing to the actors happens using dispatchers
- Actors are organized in a hierarchy via the actor system
- Actors are proxied via `ActorRef`
- Supervisor actors are used to build the fault-tolerance mechanism
- Actor path follows the URL scheme, which enables location transparency
- STM is used to provide transactional support to multiple actor state updates

More about actors

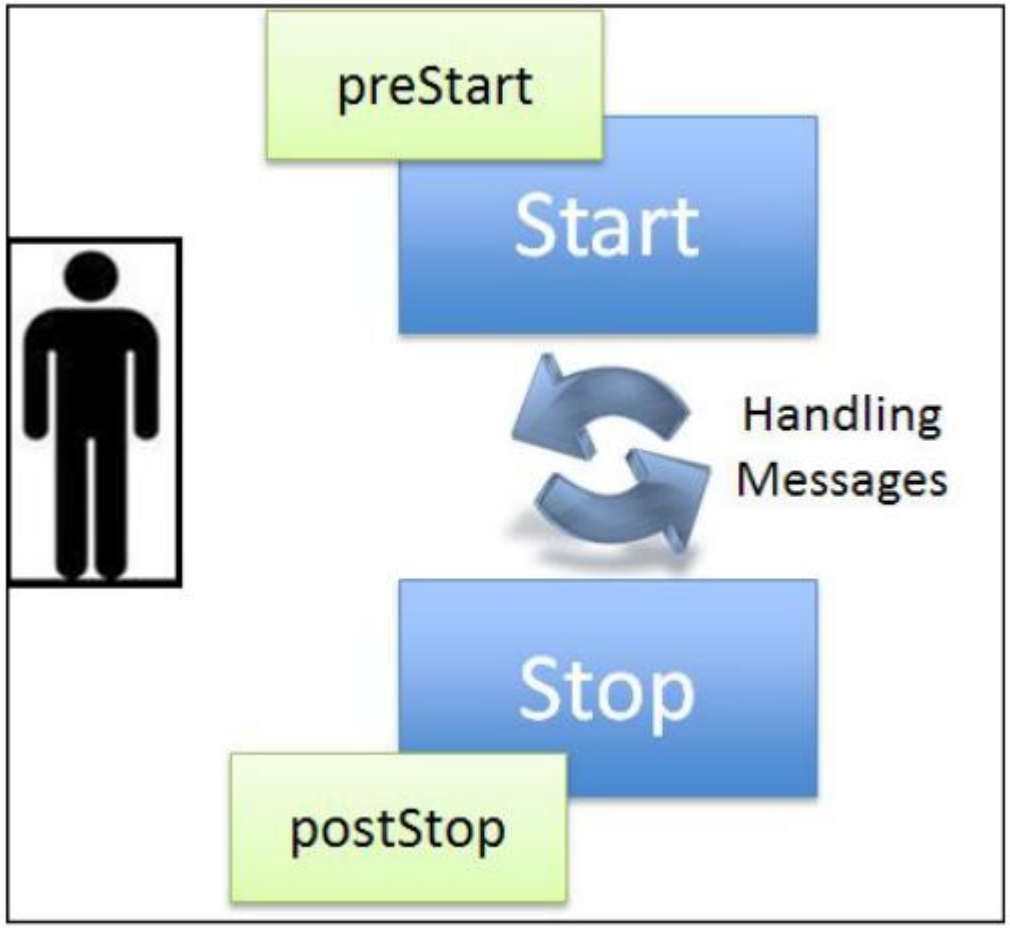
- How to create actors
- How to send/receive messages
- How to reply to messages
- How to forward messages
- How to stop actors

Actors lifecycle

- Actor is initialized and started
- Actor receives and processes messages by executing a specific behavior
- Actor stops itself when it receives a termination message

- `preStart()` and `postStop()` can be implemented to initialize/clean any resources used by the actor to process the messages
- `preRestart()` and `postRestart()` allow the actor to manage the state in case an exception has been raised and Supervisor actor restarts the actor

Actor lifecycle



Three phases

Creating Actor with default constructor

Java:

```
ActorSystem _system = ActorSystem.create("MyActorSystem");
ActorRef myActor = _system.actorOf(new Props(MyActor.class),
"myActor");
```

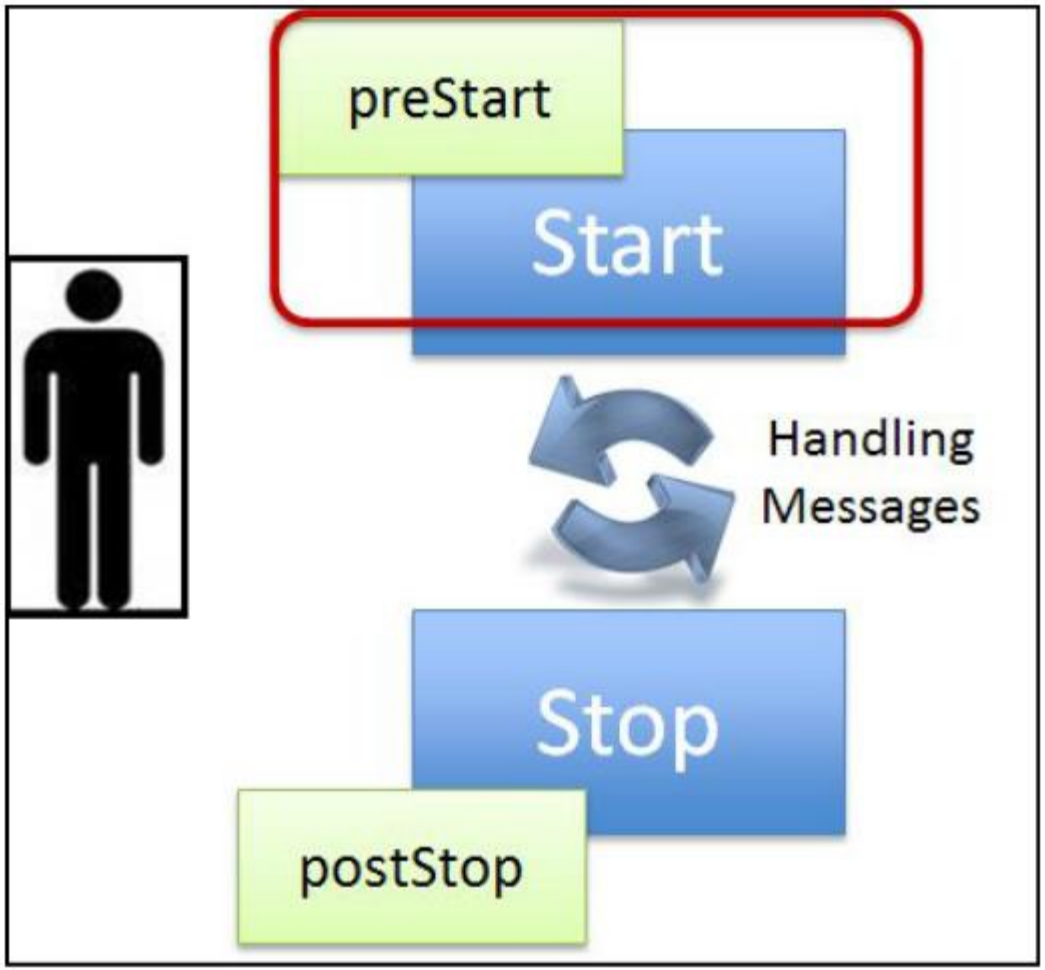
Java:

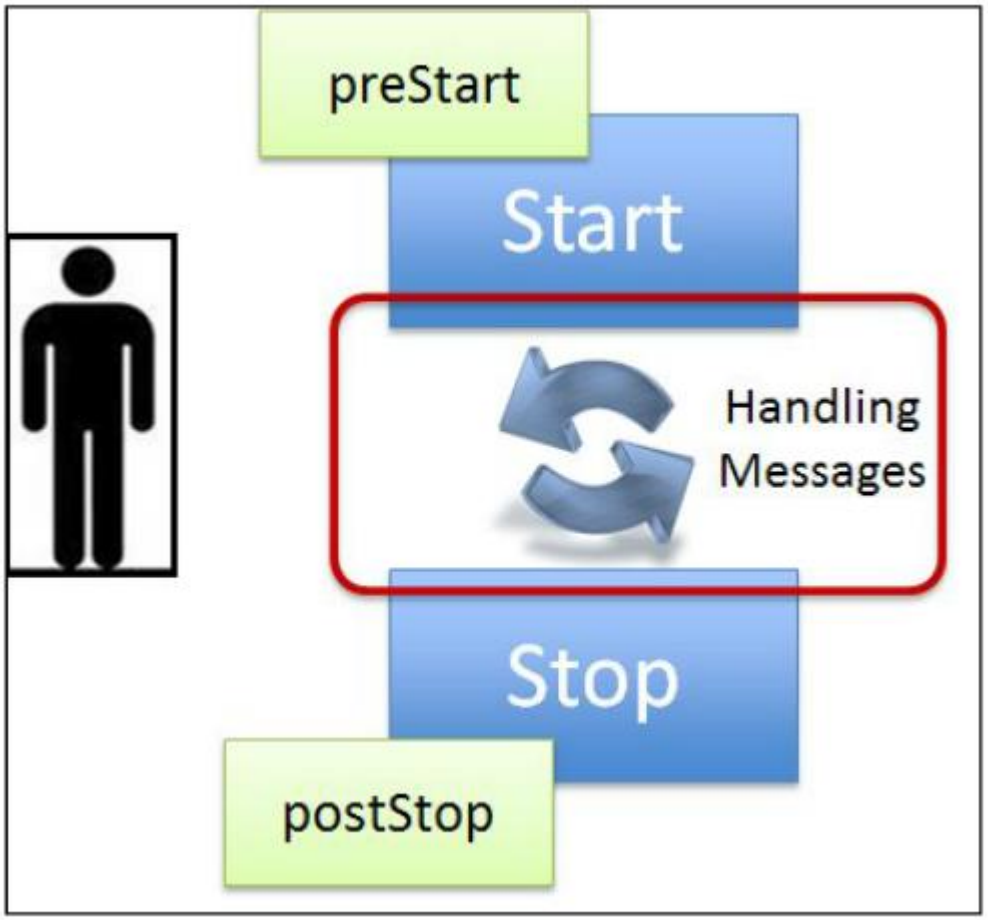
```
public class MyActor extends UntypedActor {
  public void onReceive(Object message) throws Exception {
  }
}
```

Creating an Actor within an actor hierachy

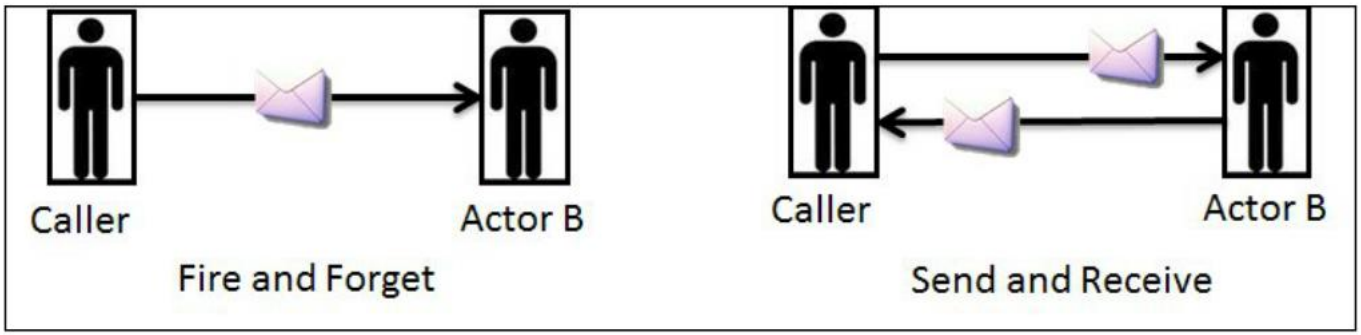
Java:

```
public class SupervisorActor extends UntypedActor {
  ActorRef myWorkerActor = getContext().actorOf(new
Props(MyWorkerActor.class), "myWorkerActor");
}
```


preStart
Start
Handling
Messages
Stop
postStop


preStart
Start
Handling
Messages
Stop
postStop

Sending Messages



`actor.tell(msg)`

`ask ()` and wiat on `future`

Immutable messages

Sending and Replying Messages

Java:

```
//explicit passing of sender actor reference
actor.tell(msg, getSelf());
//explicit passing of another actor reference
actor.tell(msg, anotherActorRef);
```

Java:

```
public void onReceive(Object request) {
    if (message instanceof String)
                    getSender().tell(message + "world");
    }
}
```

Send & Receive

```
final ArrayList<Future<Object>> futures =
                    new ArrayList<Future<Object>>();
//make concurrent calls to actors
futures.add(ask(orderActor, userId, t));
futures.add(ask(addressActor, userId, t));
```

Forward Message

Java:

```
actor.forward(message, getContext());
```

Original sender reference is forwarded

Receiving messages

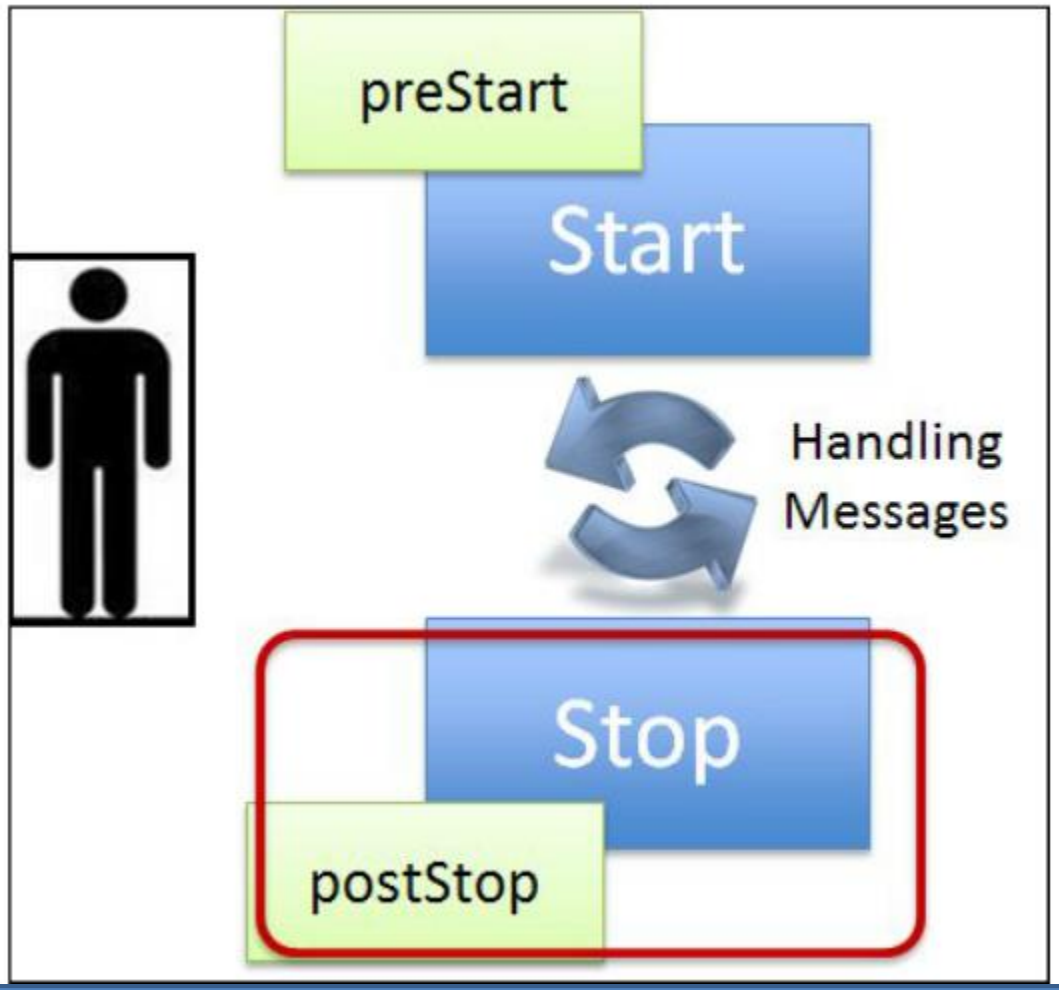
Java:

```
public class DemoActor extends UntypedActor {
  LoggingAdapter log = Logging.getLogger(
                         getContext().system(), this);
  public void onReceive(Object message) throws Exception {
    if (message instanceof String)
      log.info("Message Received by Actor ->{}", message);
    else
      unhandled(message);
  }
}
```

Replying to messages

Java:

```
public void onReceive(Object request) {
    if (message instanceof String)
                    getSender().tell(message + "world");
    }
}
```


preStart
Start
Handling
Messages
Stop
postStop

Stopping Actor

When STOP signal recieved:

1. Actor stops processing the mailbox messages.
2. Actor sends the `STOP` signal to all the children.
3. Actor waits for termination message from all its children.
4. Next, Actor starts the self-termination process that involves the following:
    - Invoking the `postStop()` method
    - Dumping the attached mailbox
    - Publishing the terminated message on `DeathWatch`
    - Informing the supervisor about self-termination

Stopping Actors

Java:

```
//first option of shutting down the actors by shutting down
//the ActorSystem
system.shutdown();
//second option of shutting down the actor by sending a
//poisonPill message
actor.tell(poisonPill());
//third option of shutting down the actor
getContext().stop(getSelf());
//or
getContext().stop(childActorRef);
```

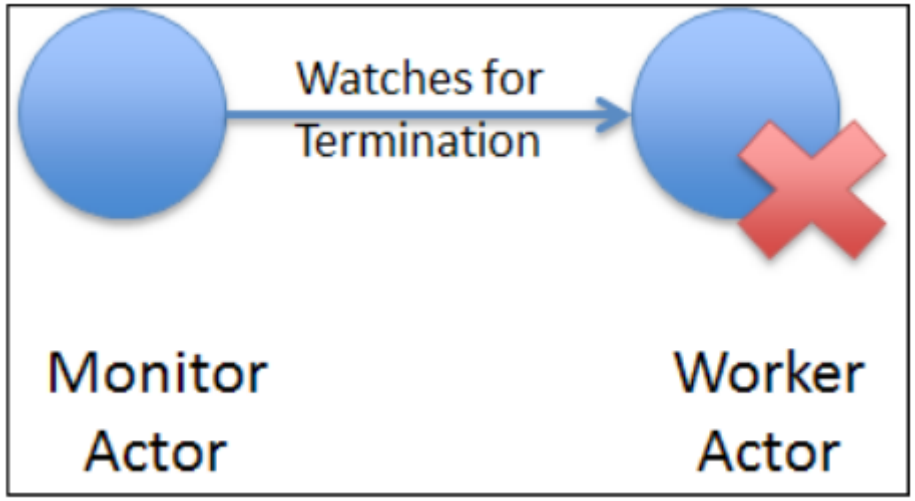
Killing Actor

An actor can be killed when a `kill()` message is sent to it. Unlike `PoisonPill`, which is an asynchronous way to shut down the actor, `kill()` is a synchronous way. The killed actor sends `ActorKilledException` to its parent.

Java:

```
actor.tell(kill());
```

Lifecycle Monitoring



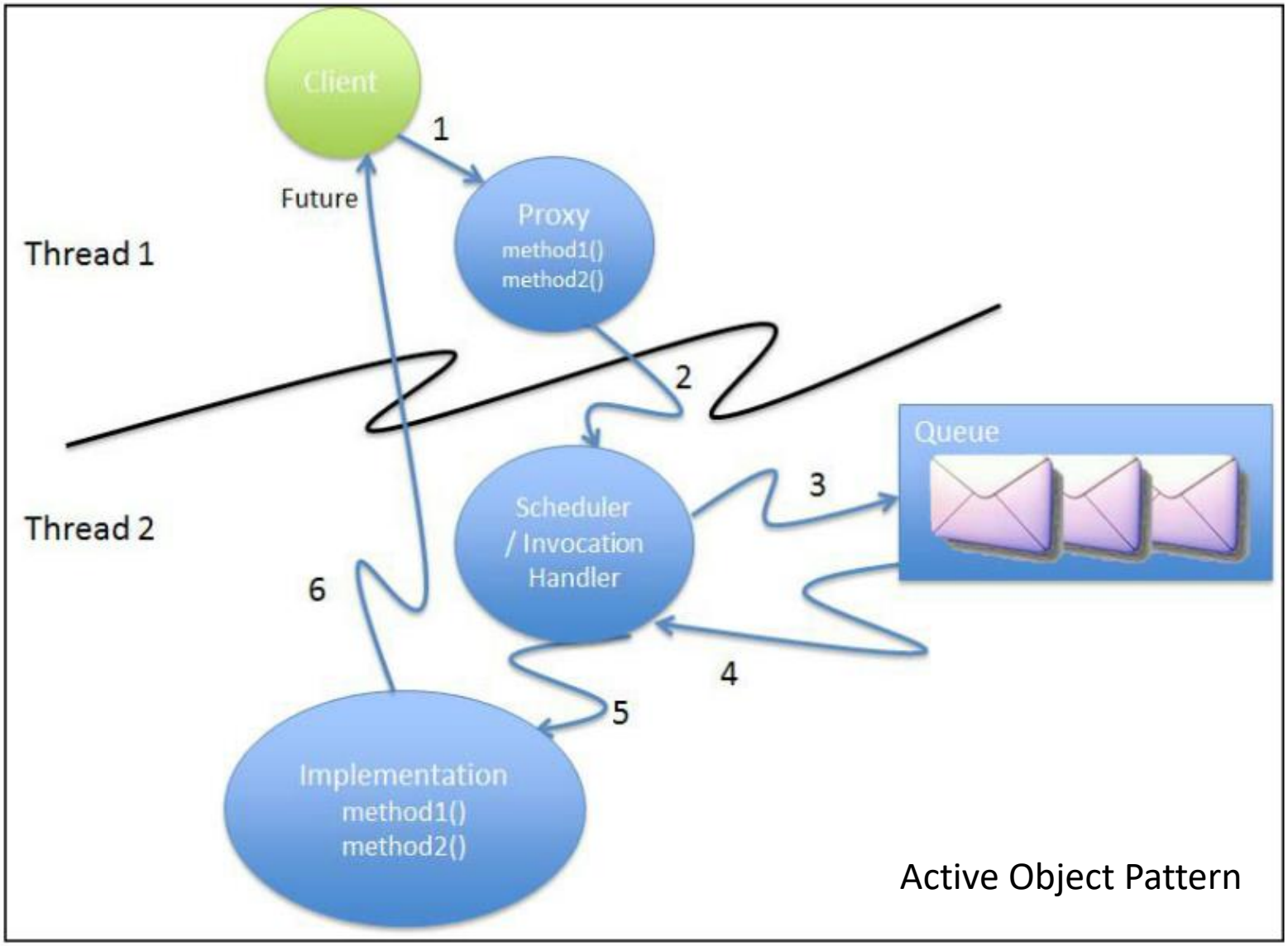
Typed Actors

They implement not only System interface but their on Business Interface.

Java:

```
public interface CalculatorInt {

    public Future<Integer> add(Integer first, Integer second);

    public Future<Integer> subtract(Integer first,
                                    Integer second);

    public void incrementCount();

    public Option<Integer> incrementAndReturn();

}
```

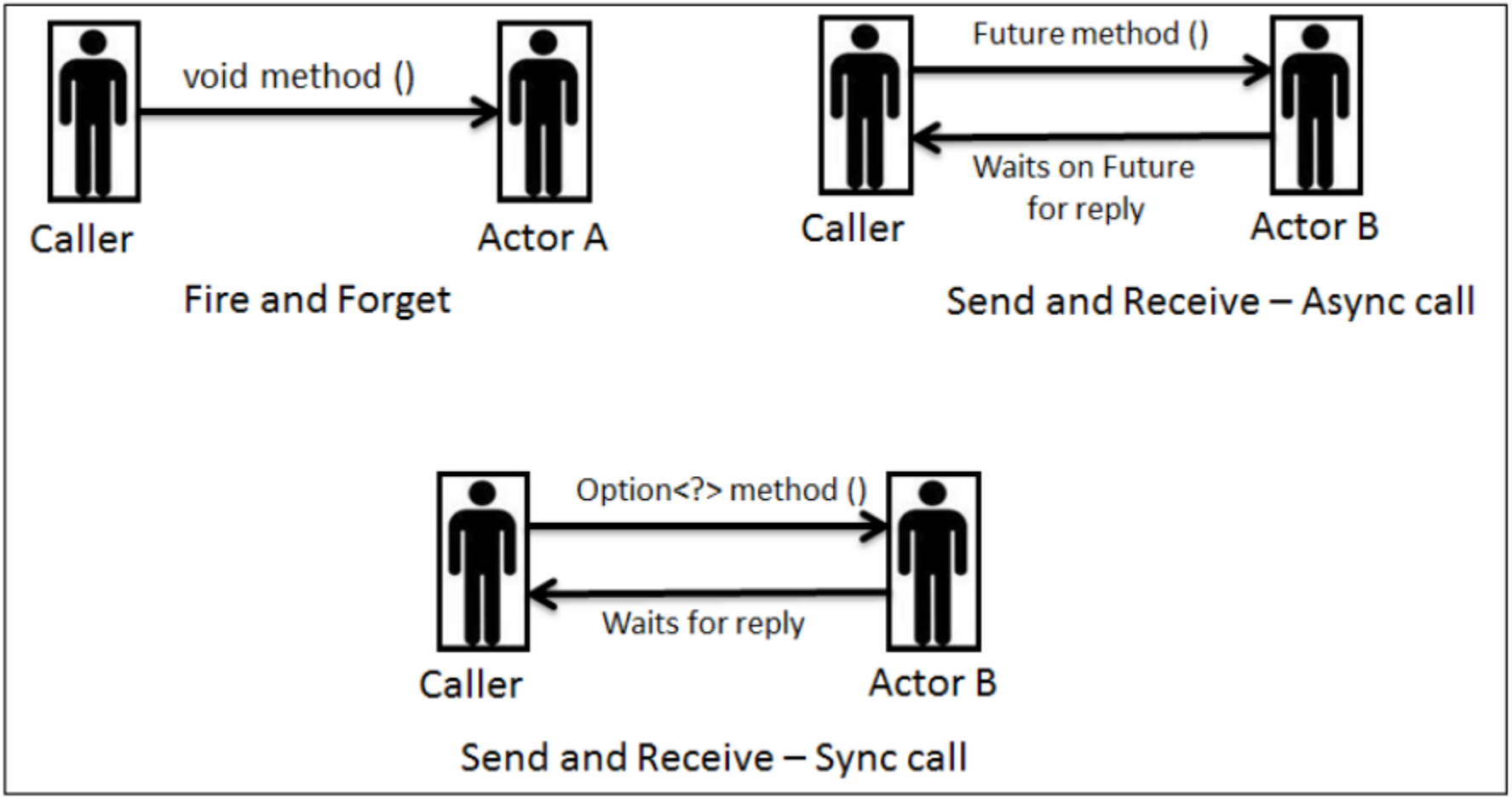
Decopule Invocation and Execution



Active Object Pattern

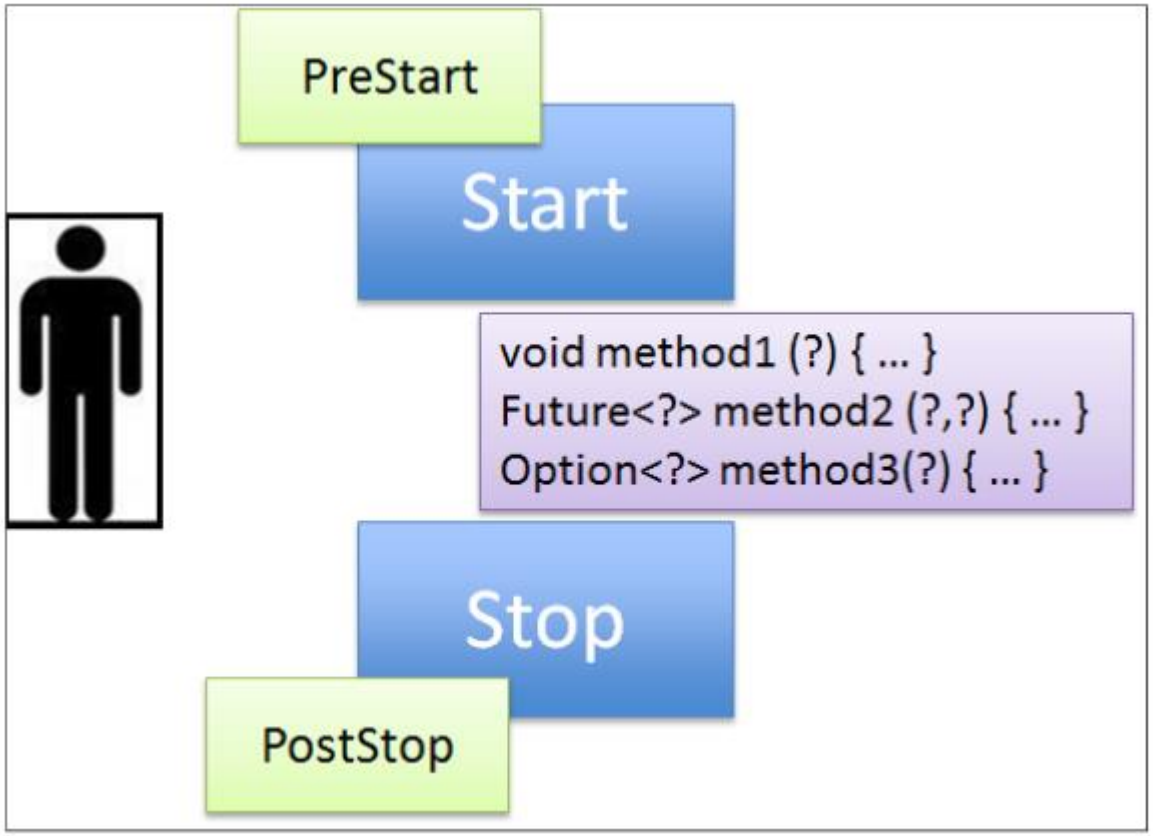
1. At runtime, the client calls the proxy object and invokes the method.
2. The proxy in turn passes the method calls as method requests to a scheduler or invocation handler that intercepts the call.
3. Scheduler or invocation handler enqueues the method requests on a queue.
4. The scheduler continuously monitors the queue and determines which method request(s) have become runnable, that is when their synchronization constraints are met. At this point, the scheduler or invocation handler dequeues the requests from the queue.
5. Scheduler or invocation handler dispatches the requests to the implementation object.
6. The implementation object, running on the same thread as the scheduler, processes the request and returns any value to the client as `Future`.

Sending Messages



Sending Receiving

```
Future<Integer> future = calculator.add(Integer.valueOf(14),
                  Integer.valueOf(6));
Integer result = Await.result(future, timeout.duration());
```


PreStart
Start
void method1 (?) { ... }
Future<?> method2 (?,?) { ... }
Option<?> method3(?) { ... }
Stop
PostStop

Dispatchers

Dispatchers are the communiction coordinators that are responsible for receiving and passing messages.

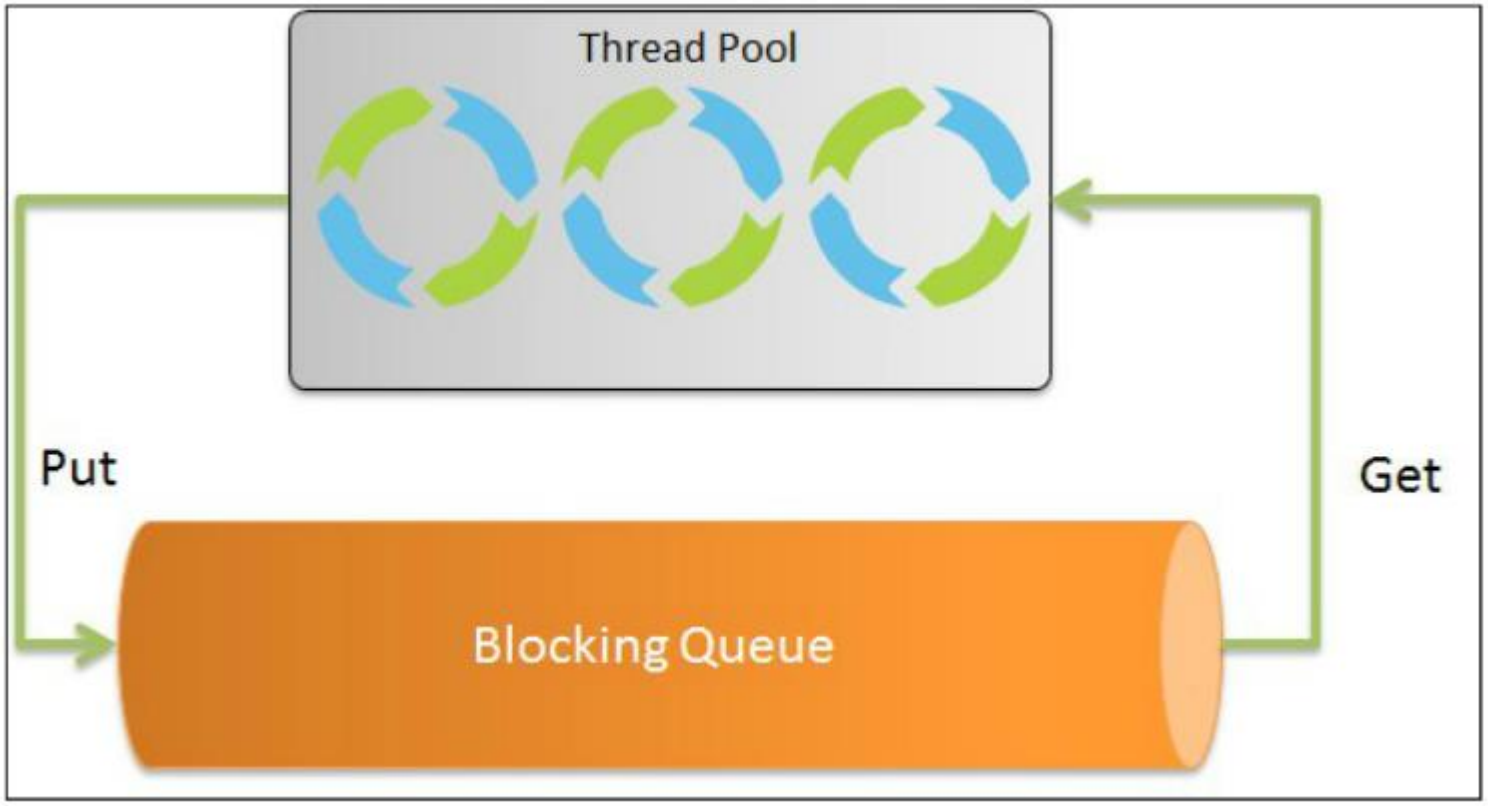
Dispatchers as a Pattern

- **Centralized control**: Dispatchers provide a central place from where various messages/requests are dispatched. The word "centralized" means code is re-used, leading to improved maintainability and reduced duplication of code.
- **Application partitioning**: There is a clear separation between the business logic and display logic. There is no need to intermingle business logic with the display logic.
- **Reduced inter-dependencies**: Separation of the display logic from the business logic means there are reduced inter-dependencies between the two. Reduced inter-dependencies mean less contention on the same resources, leading to a scalable model.

Producer-Consumer model

The threads that submit task is different that the threads execute the task.

ForkJoinPool

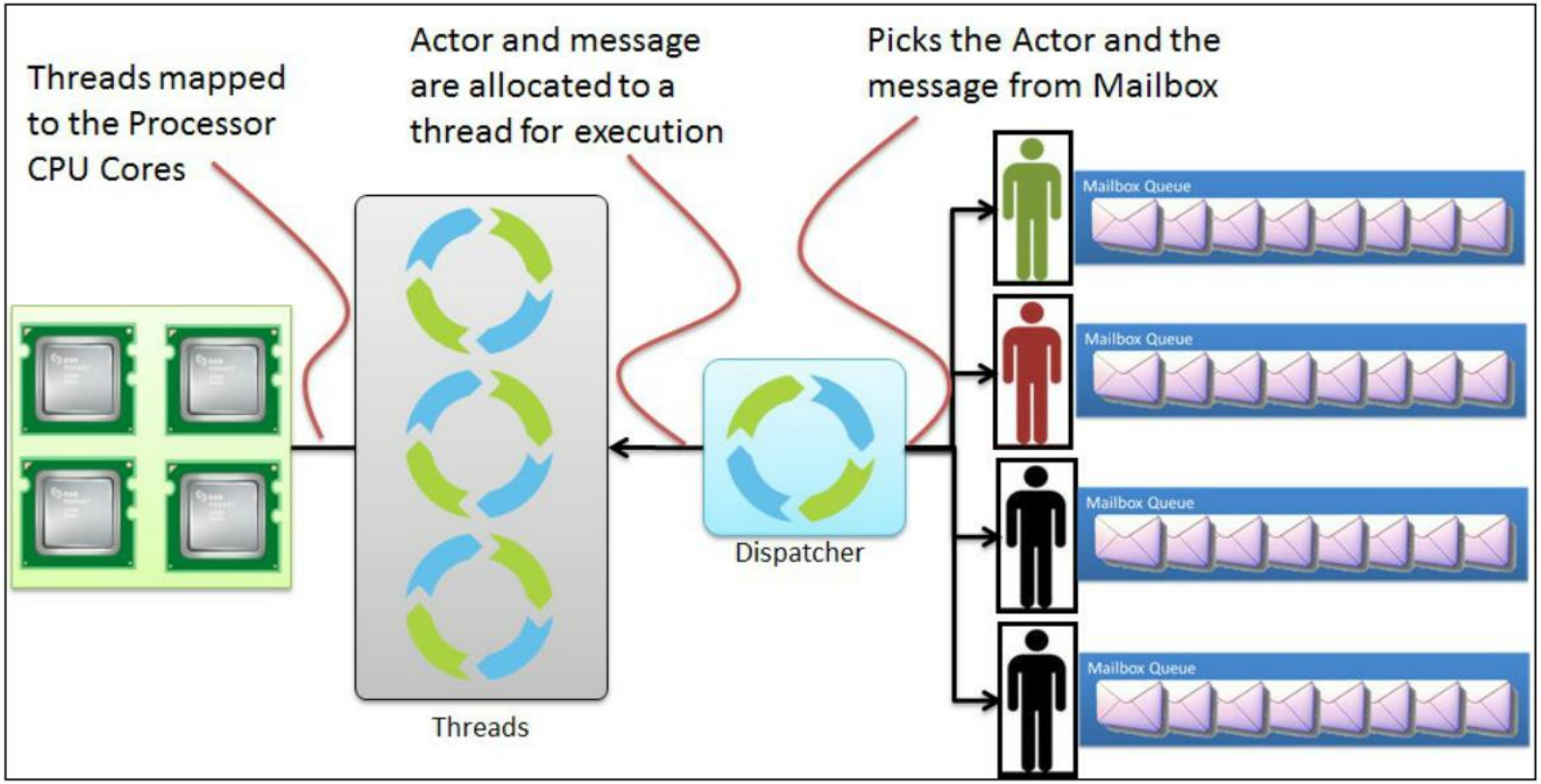


Message Dispatchers

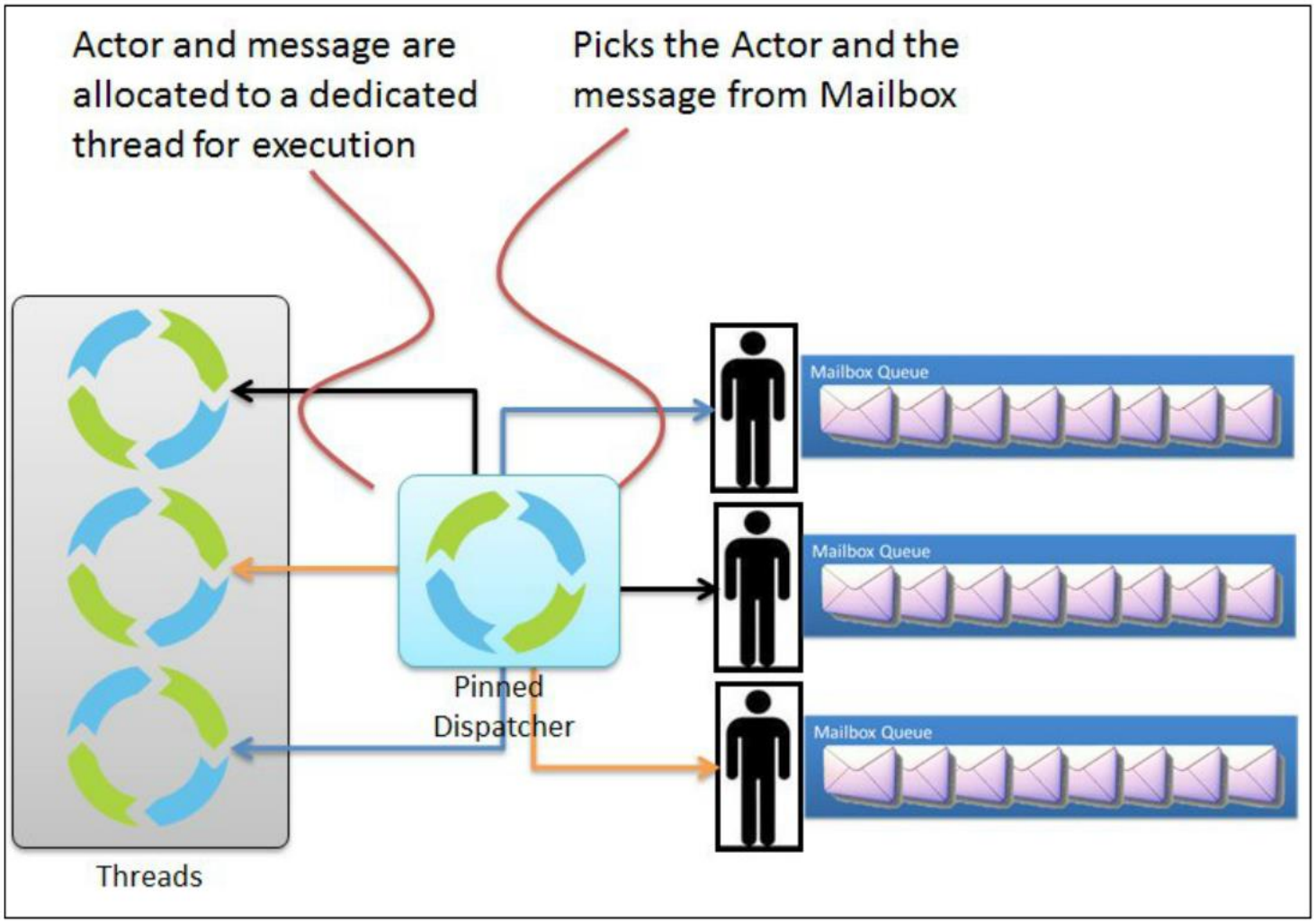
There are four different types of message dispatchers:

1. Thread-based
2. Event-based
3. Priority event-based
4. Work-stealing

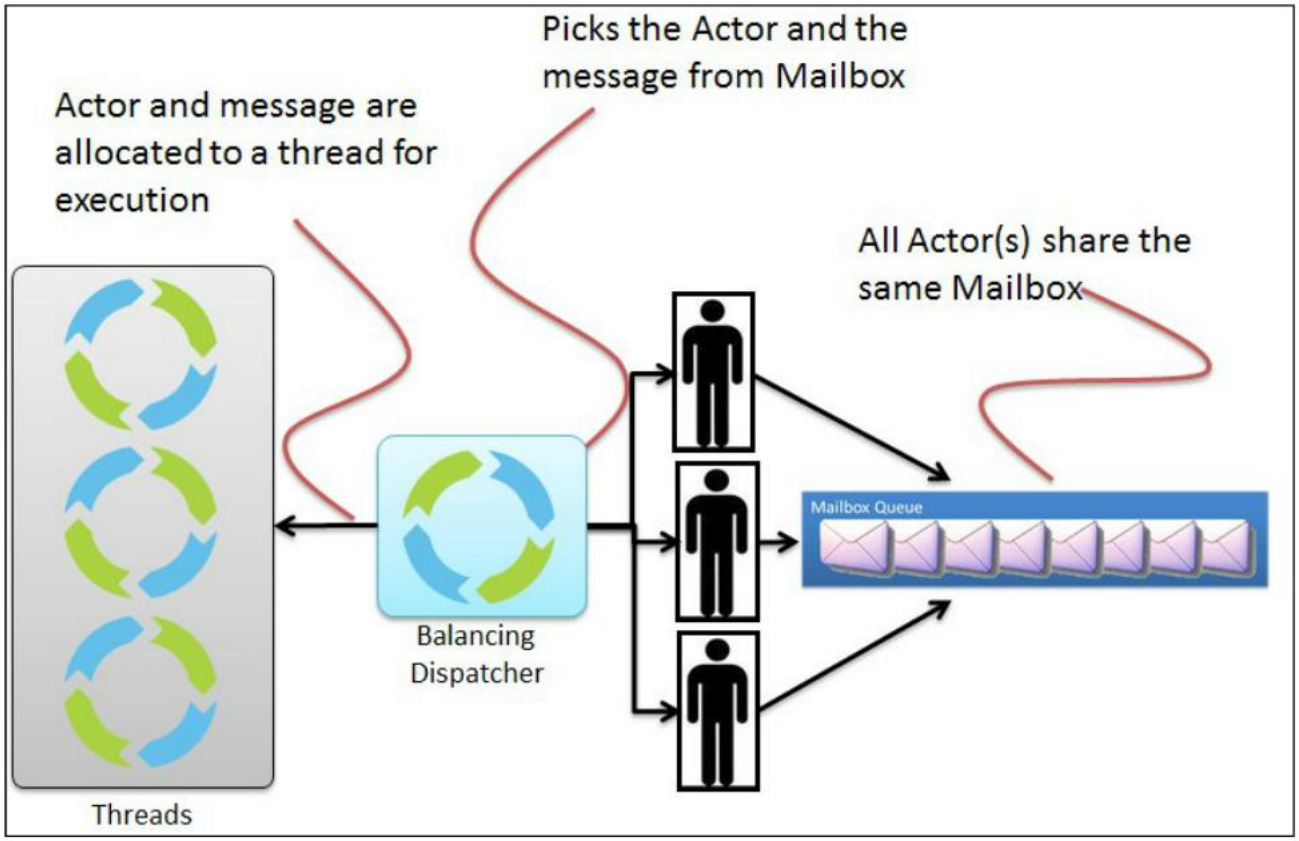
Event-based Dispatcher



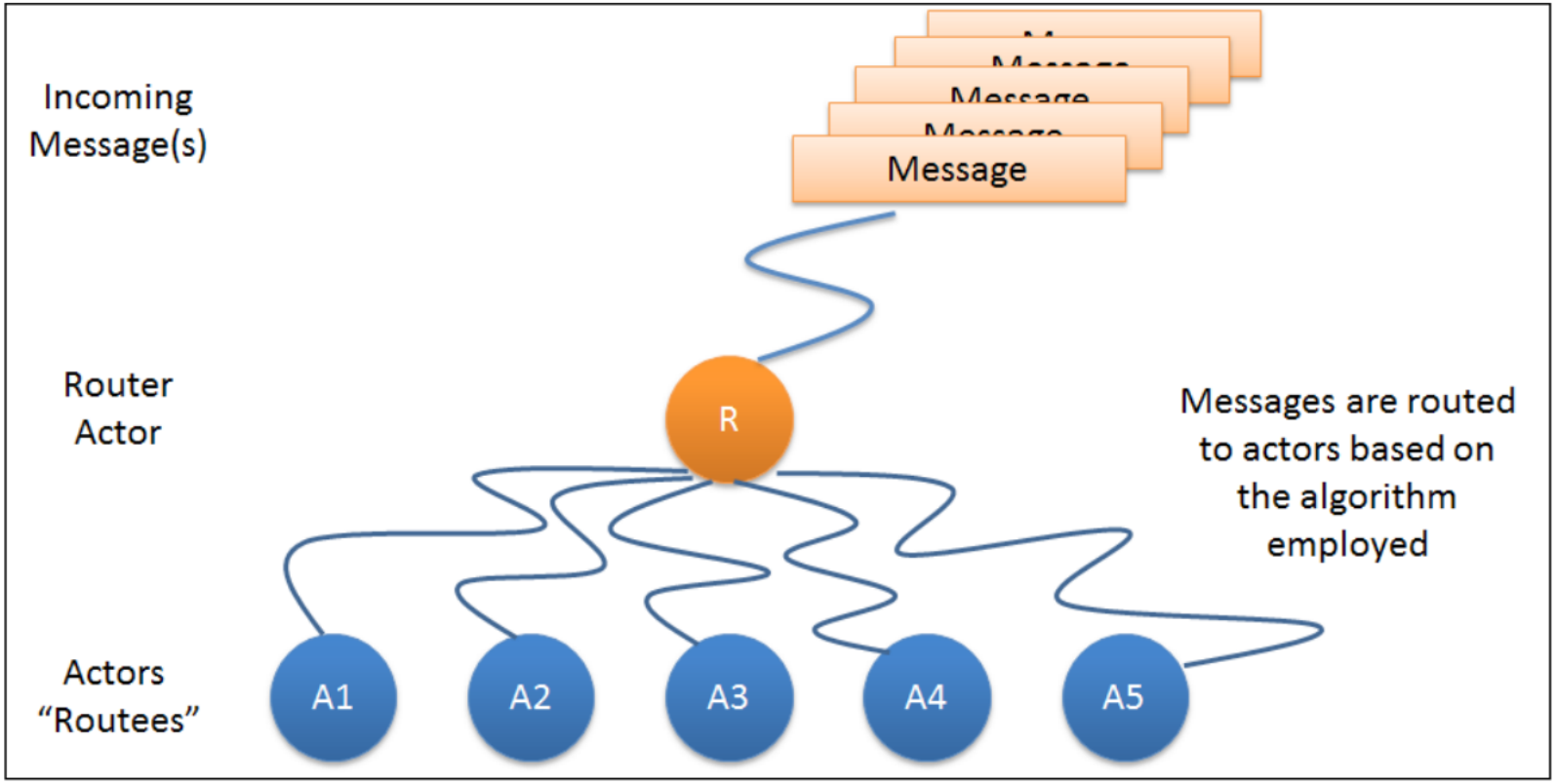
Pinned Dispatcher



Balancing Dispatcher



Routers


Incoming
Message(s)
Message
Router
Actor
R
Messages are routed
to actors based on
the algorithm
employed
Actors
"Routees"
A1
A2
A3
A4
A5

Akka Routers

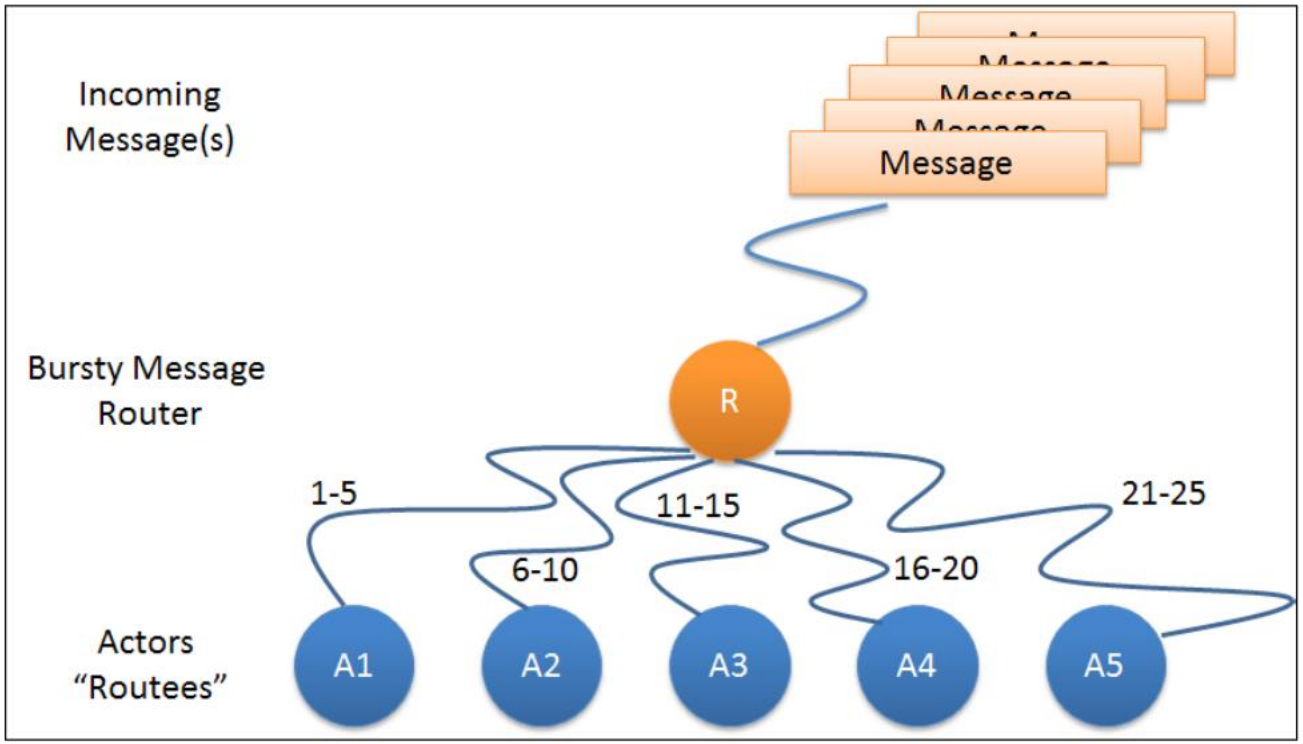
- **Round robin router**: It routes the incoming messages in a circular order to all its routees
- **Random router**: It randomly selects a routee and routes the message to the same
- **Smallest mailbox router**: It identifies the actor with the least number of messages in its mailbox and routes the message to the same
- **Broadcast router**: It forwards the same message to all the routees
- **Scatter gather first completed router**: It forwards the message to all its routees as a `future`, then whichever routee actor responds back, it takes the results and sends them back to the caller

Round Robin Router

Java:

```
ActorRef router = system.actorOf(new Props
(MyActor.class).withRouter(new RoundRobinRouter
(nrOfInstances)));
```

Custom Router

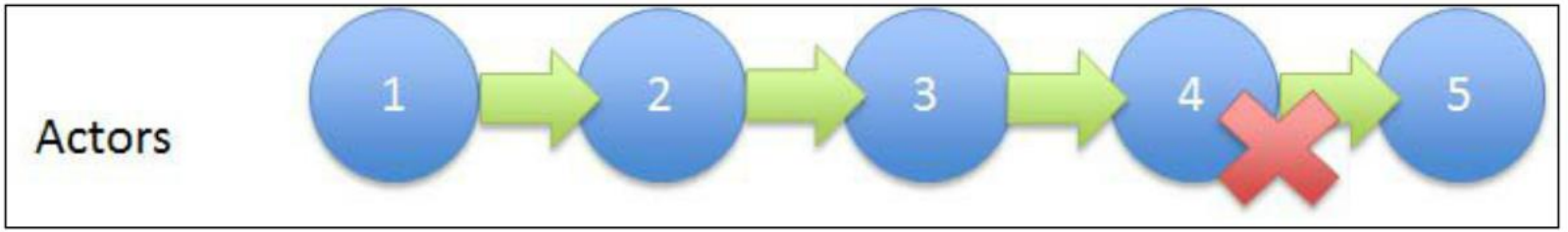


Router with Distributed Actors

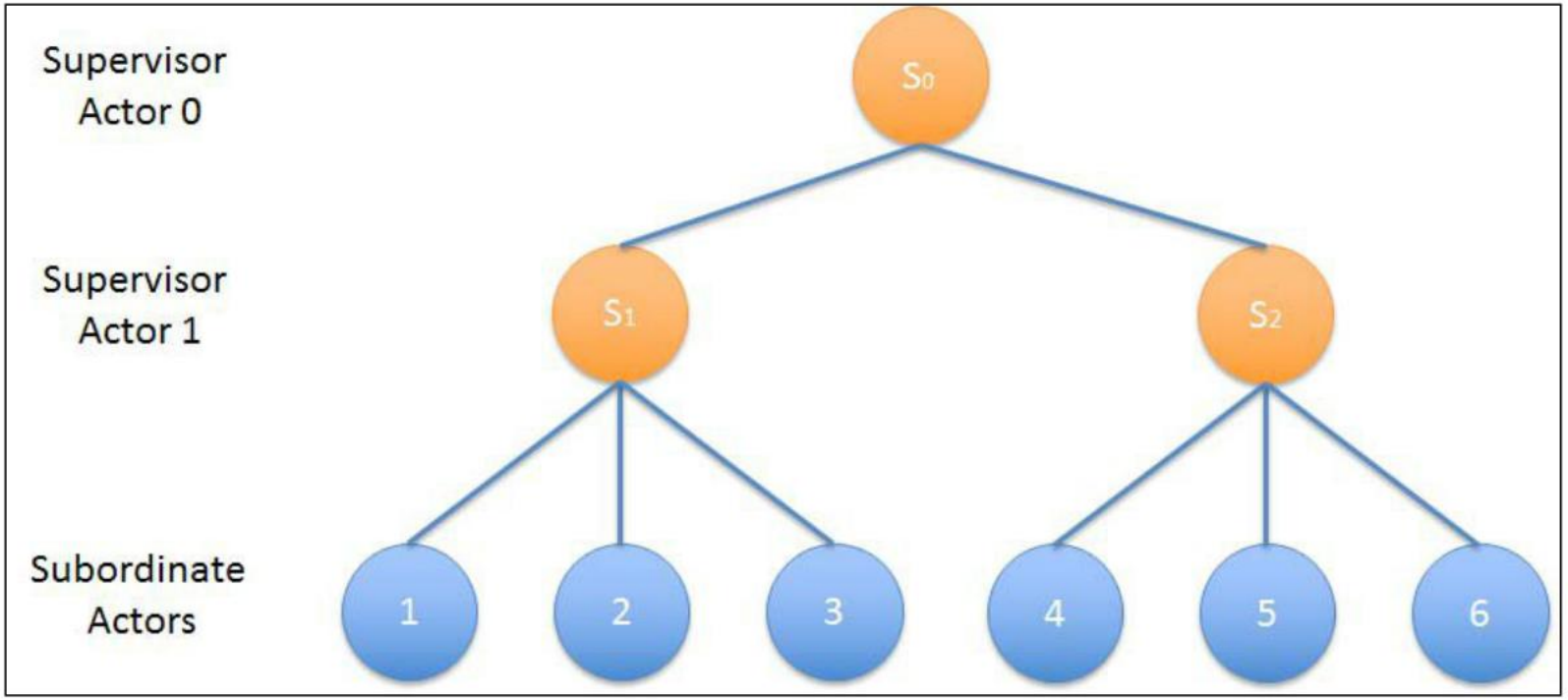
Java:

```
Address addr1 = new Address("akka", "remotesys", "host1", 1234);
Address addr2 = new Address("akka", "remotesys", "host2", 1234);

Address[] addresses = new Address[] { addr1, addr2 };

ActorRef routerRemote = system.actorOf(new Props(MyEchoActor.class)
  .withRouter(new RemoteRouterConfig(new RoundRobinRouter(5),
addresses)));
```

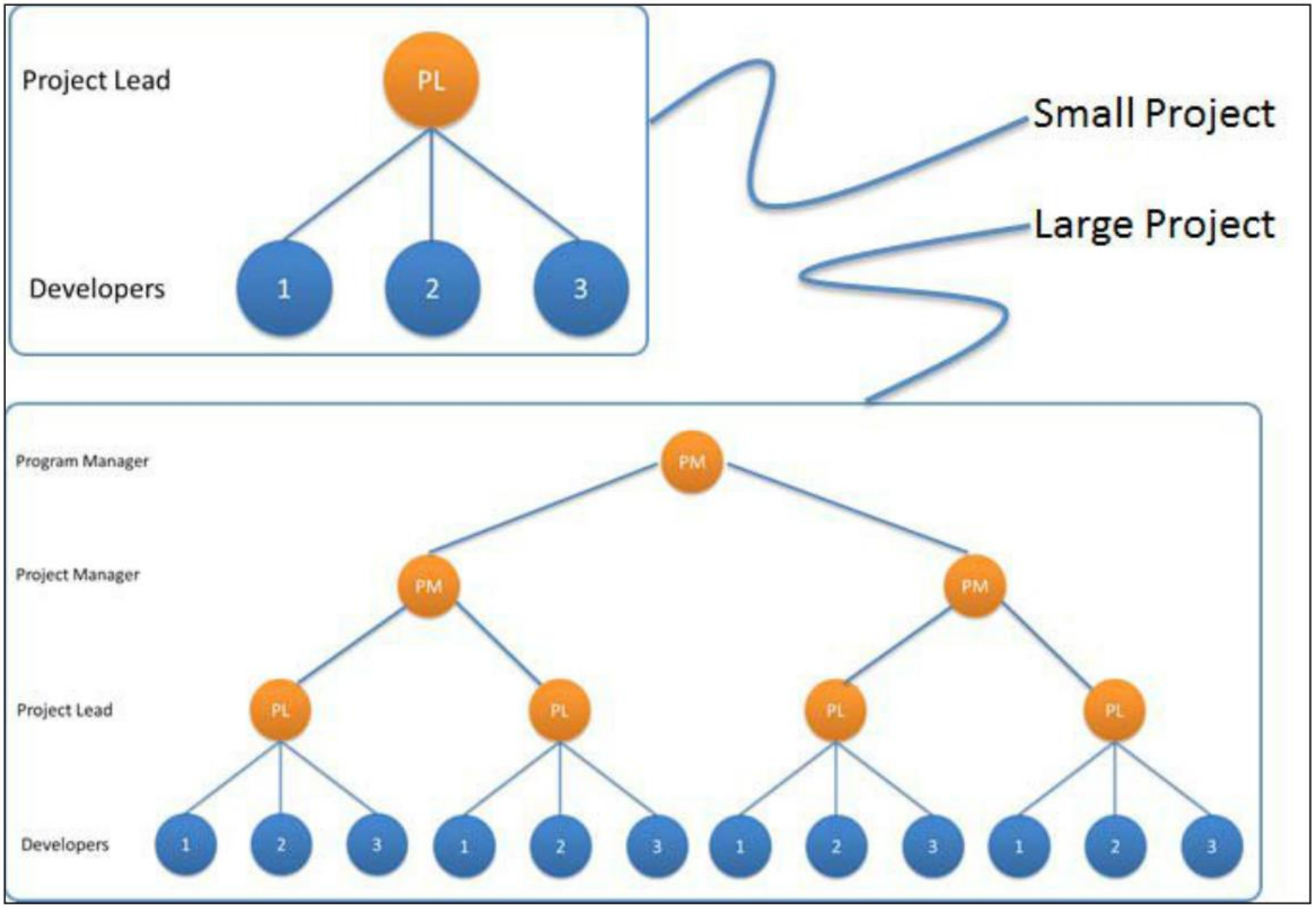
Actors Crash



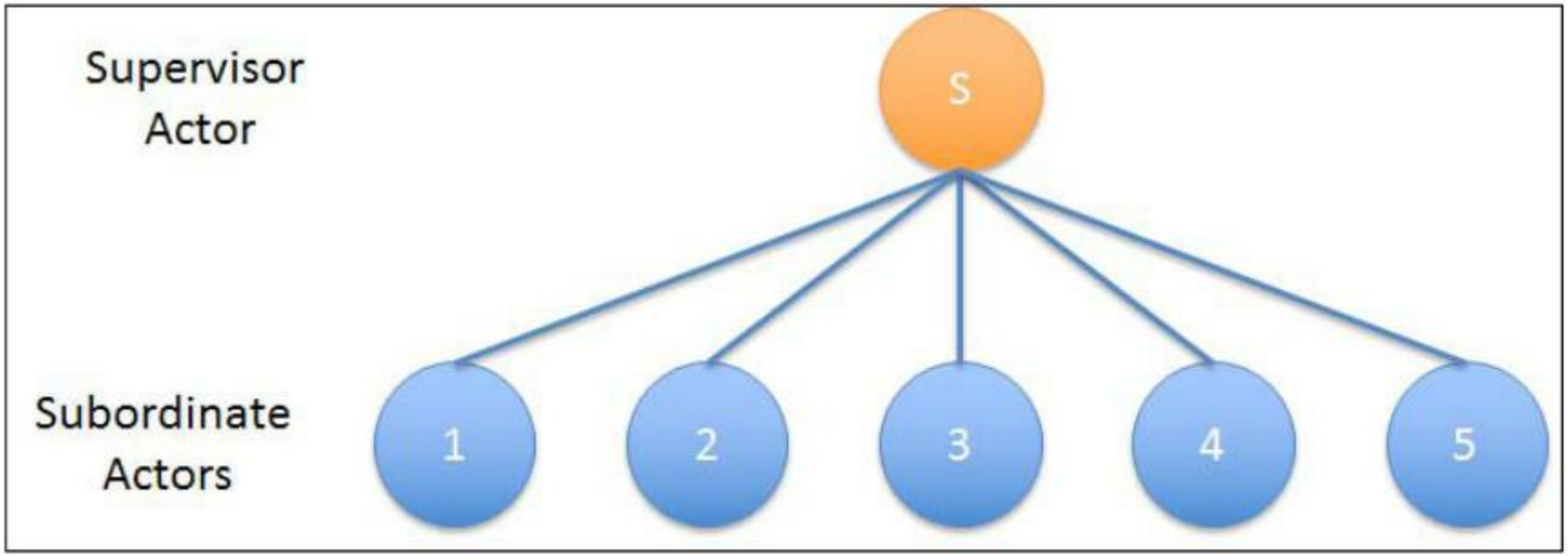
Supervision



Lager System

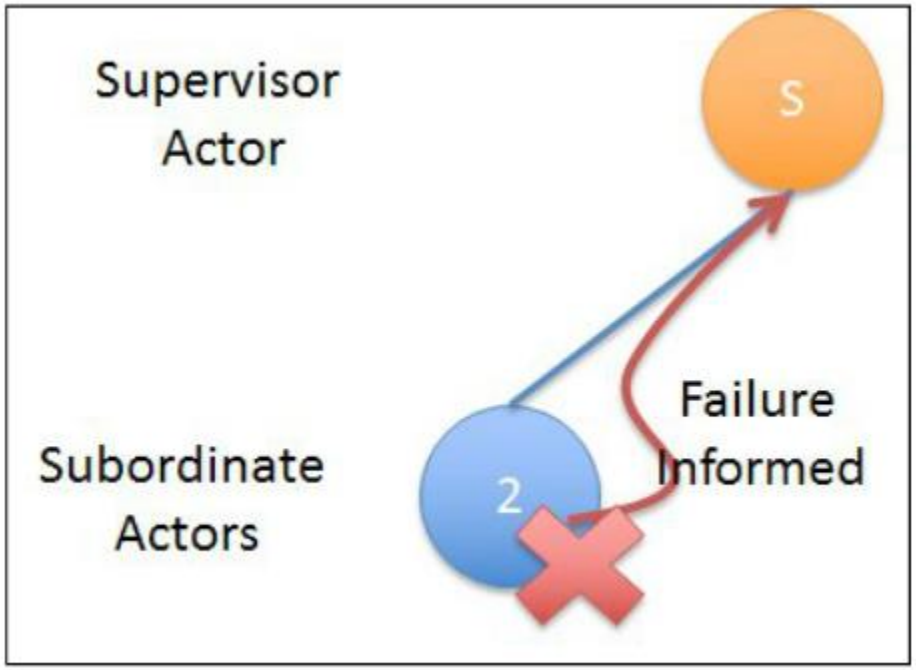


Supervisor

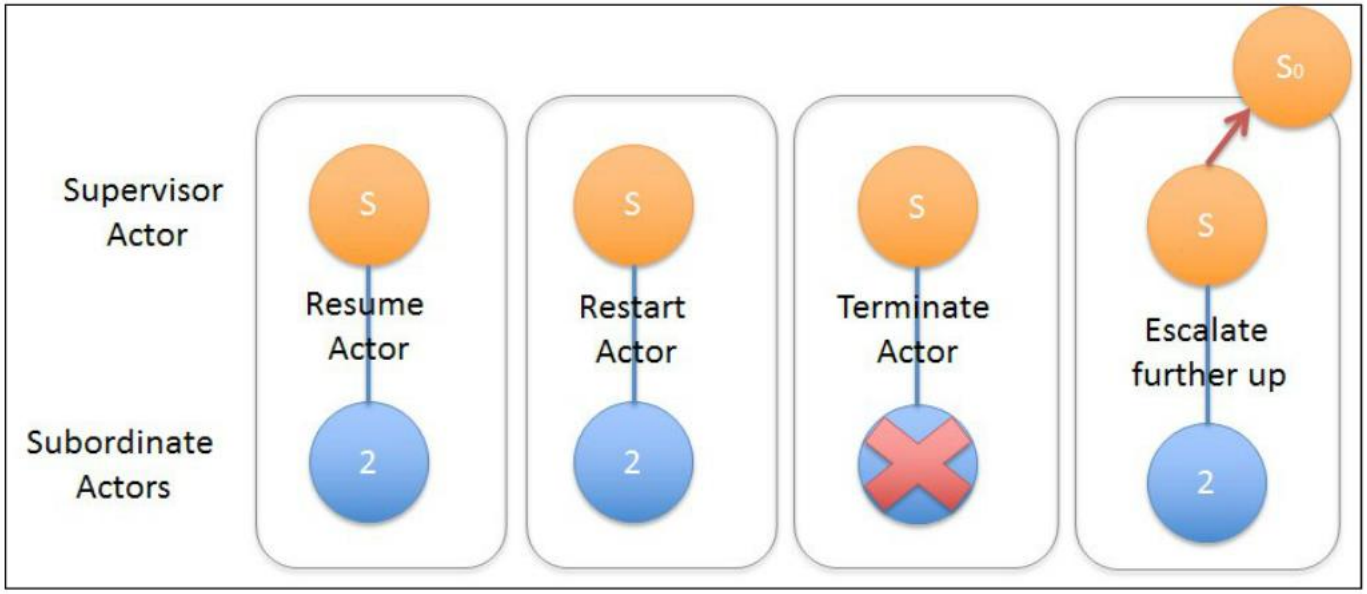


Supervisor delegate the tasks to subordinate Actors and manges their lifcycle

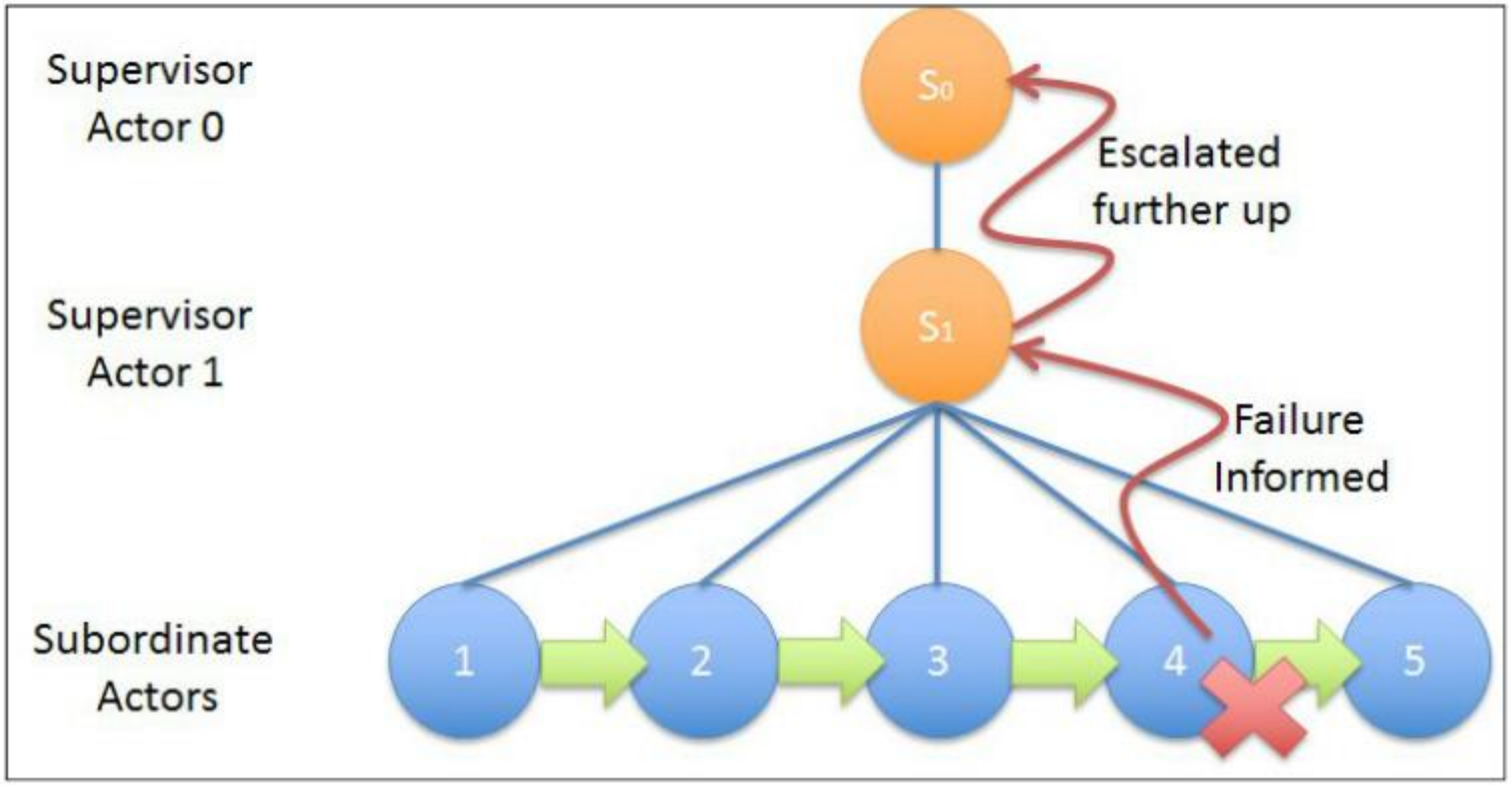
Supervisor Handle Fauilure



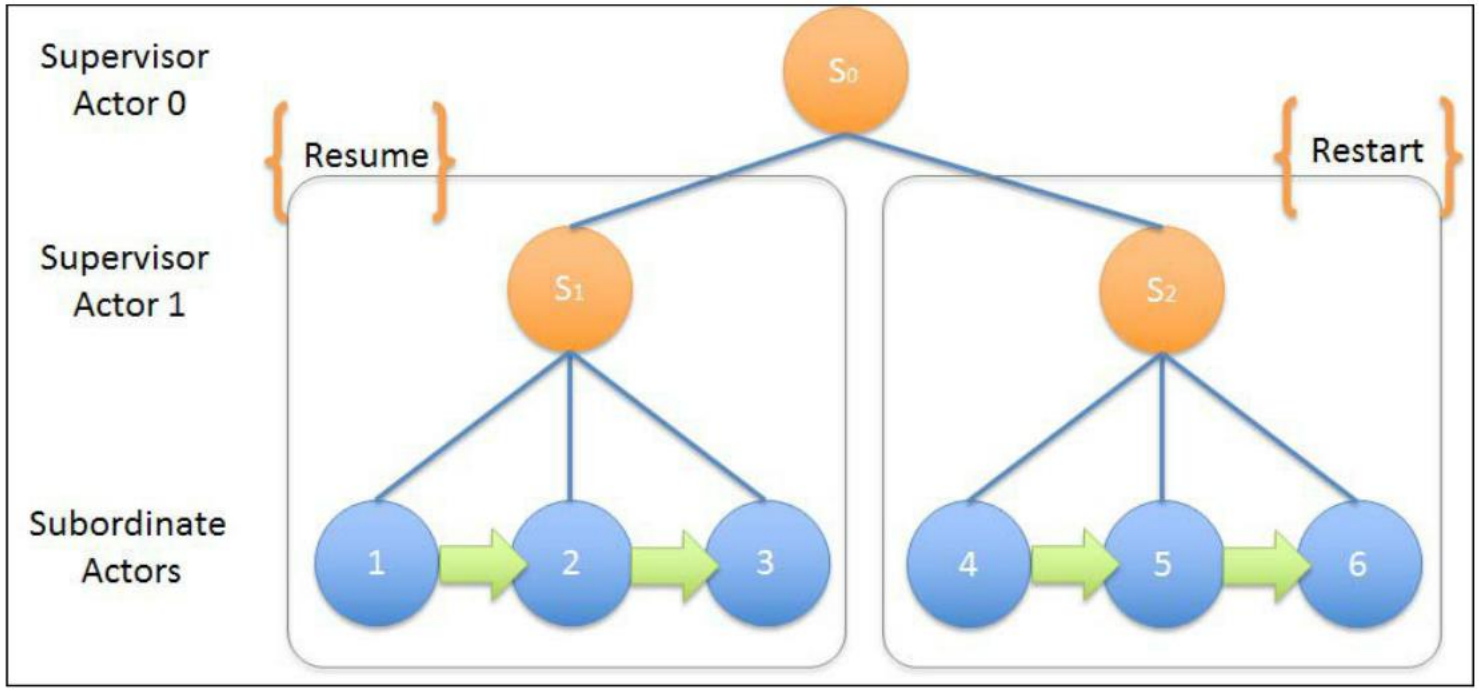
Supevisor Actions



Failure Escalation

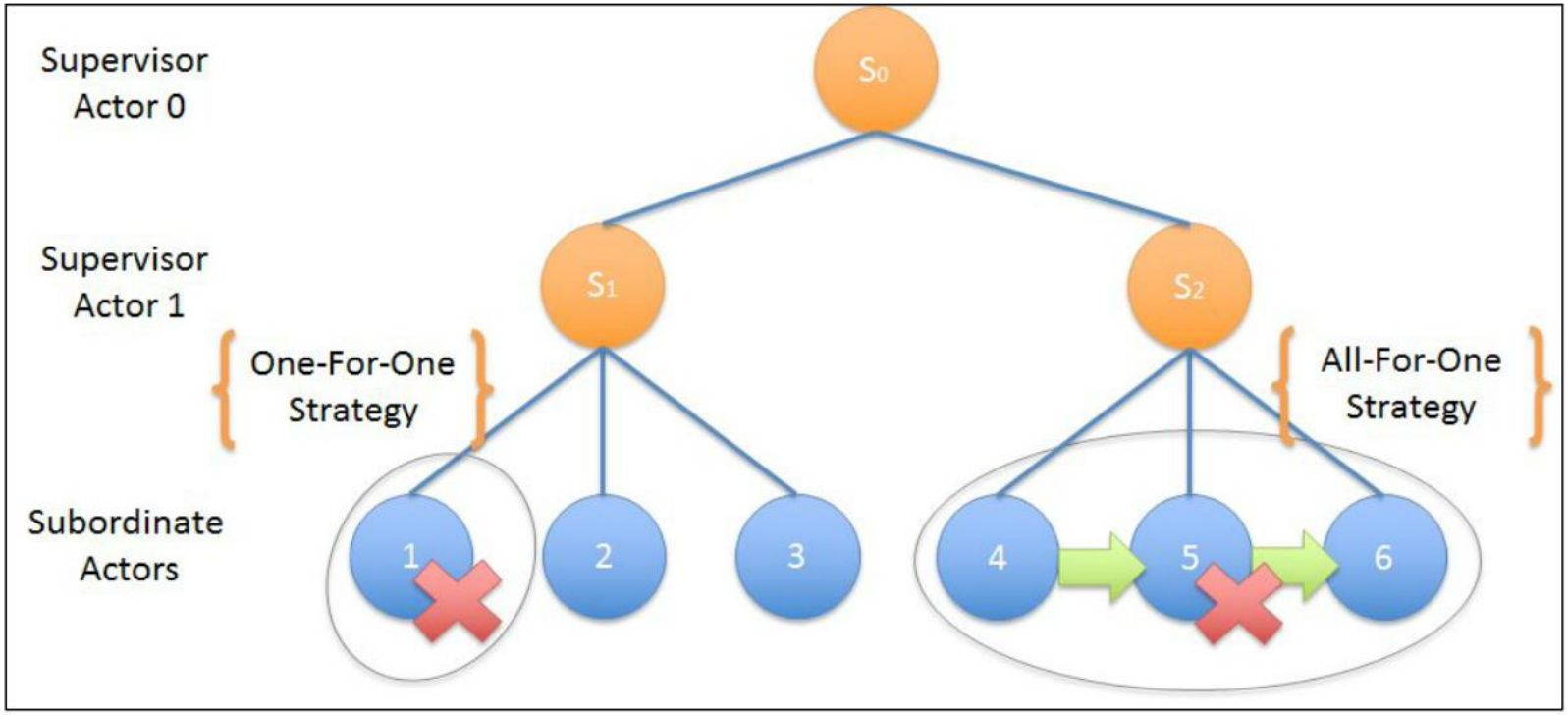


Resume vs. Restart

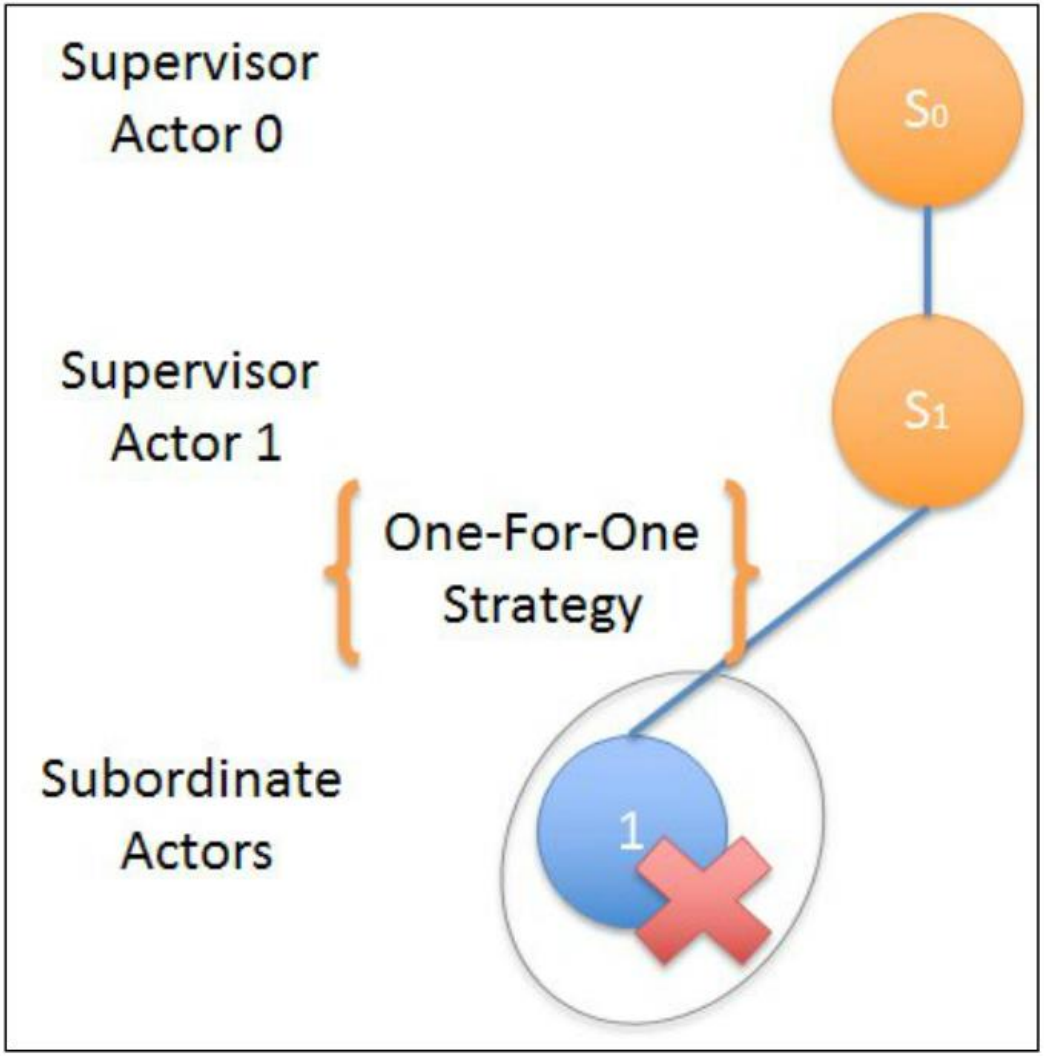


Start from initial state

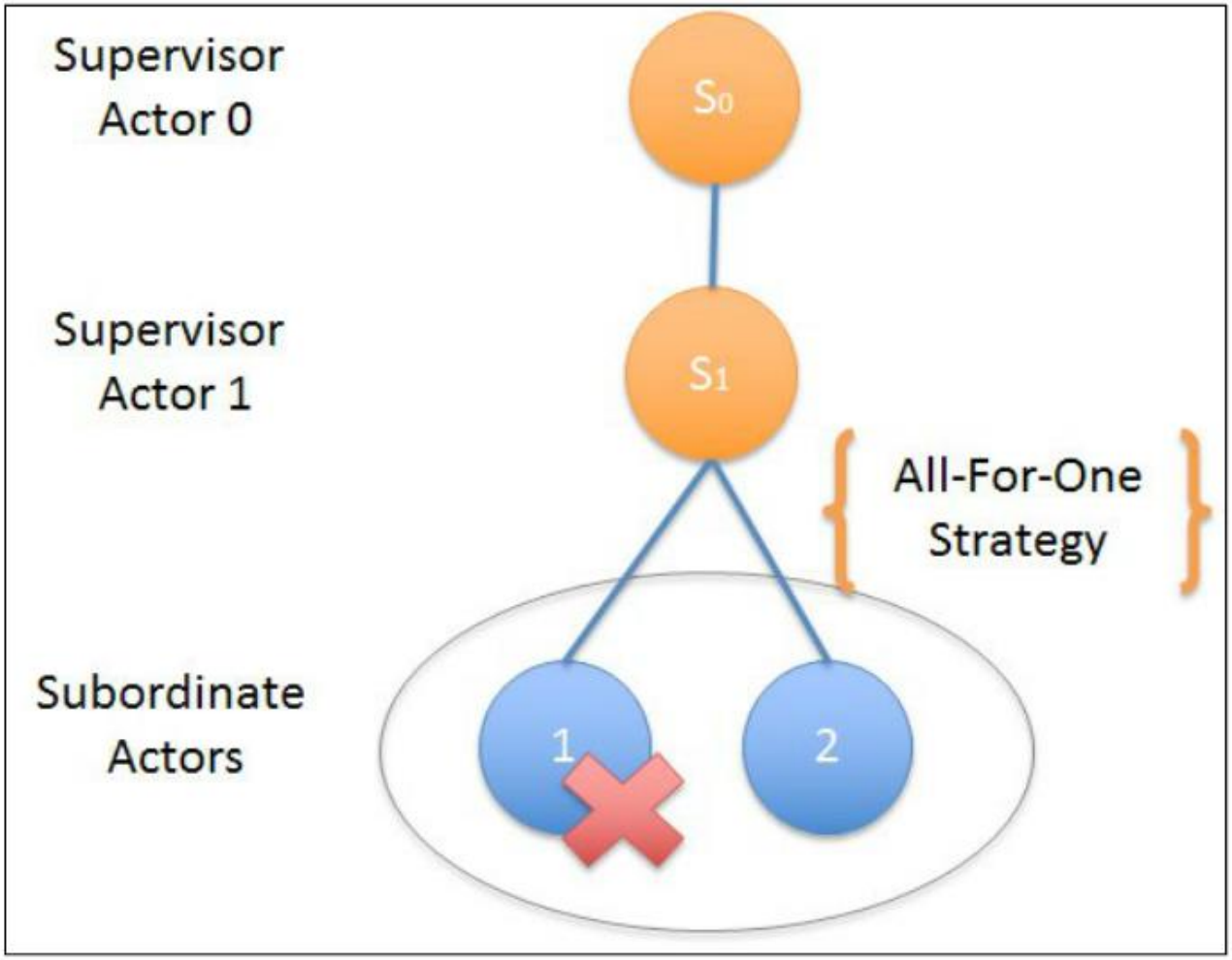
Supervisor Strategy



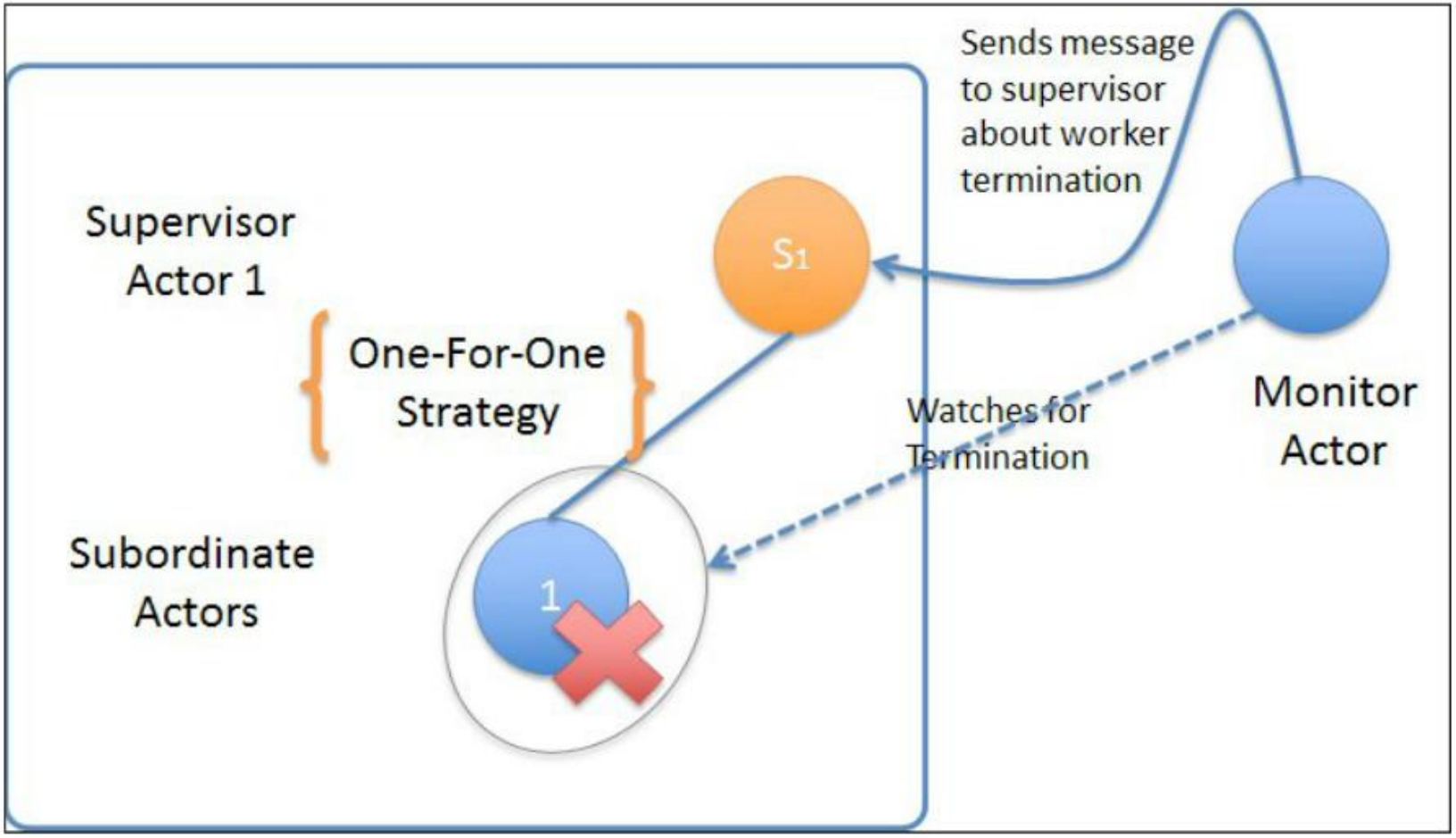
One-For-One


Supervisor
Actor 0
S0
Supervisor
Actor 1
S1
One-For-One
Strategy
Subordinate
Actors
1

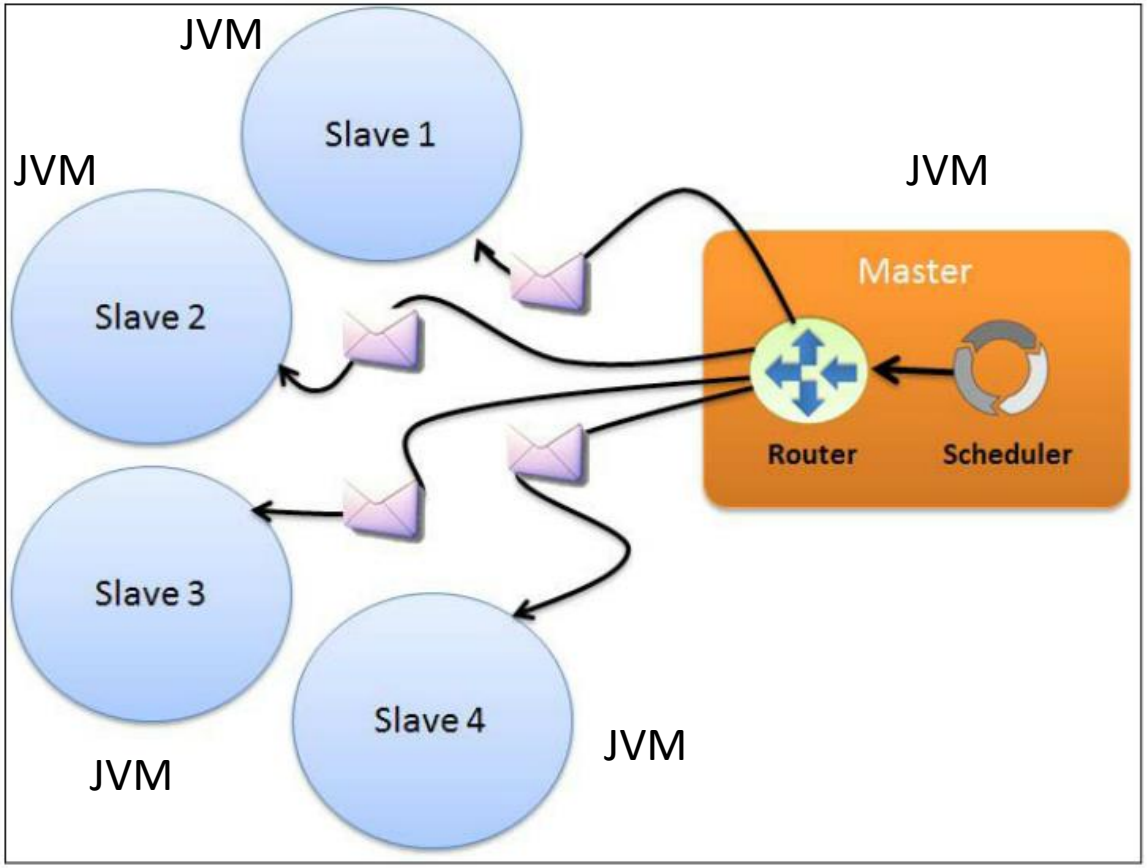
All-For-One


Supervisor
Actor 0
S0
Supervisor
Actor 1
S1
All-For-One
Strategy
Subordinate
Actors
1
2

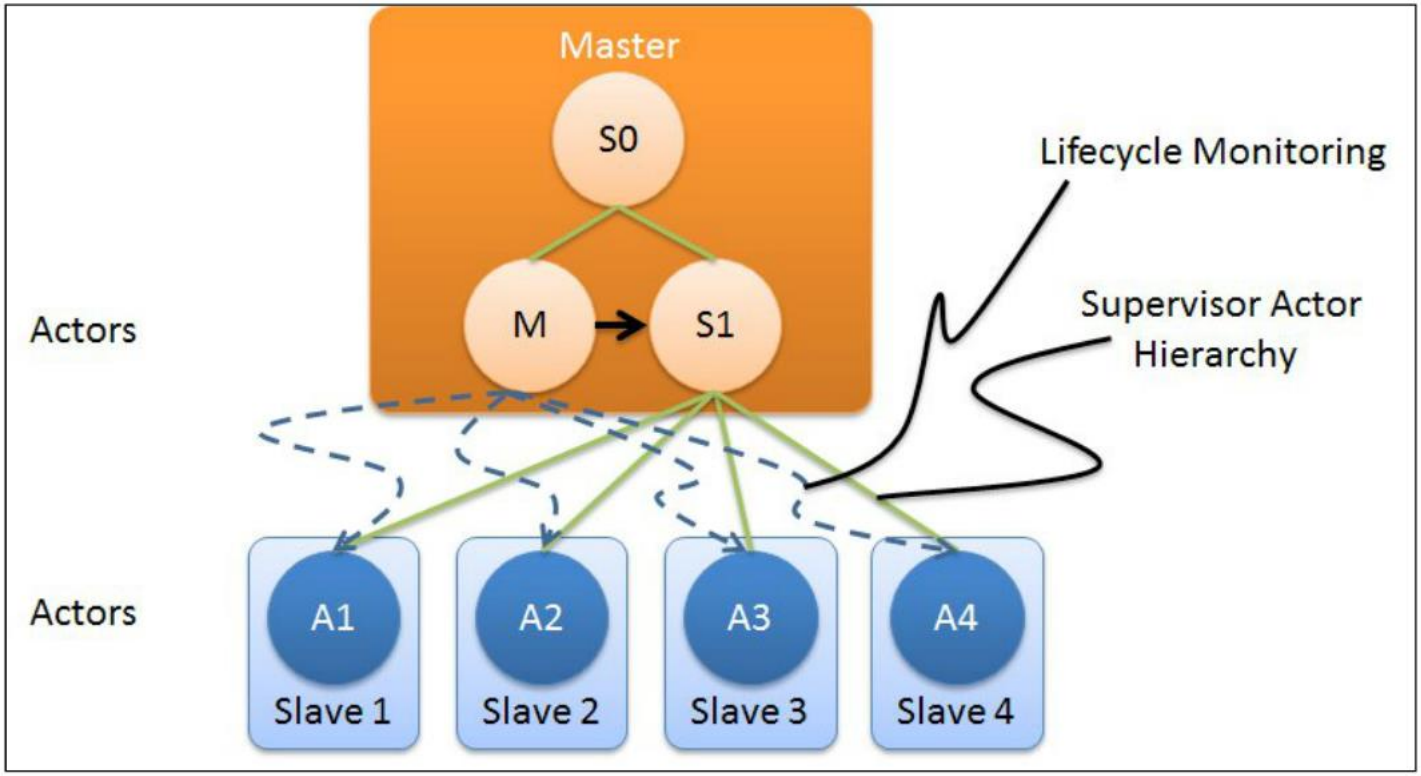
Lifecycle Monitoring



Master Slave Ex.



Lifecycle Monitoring



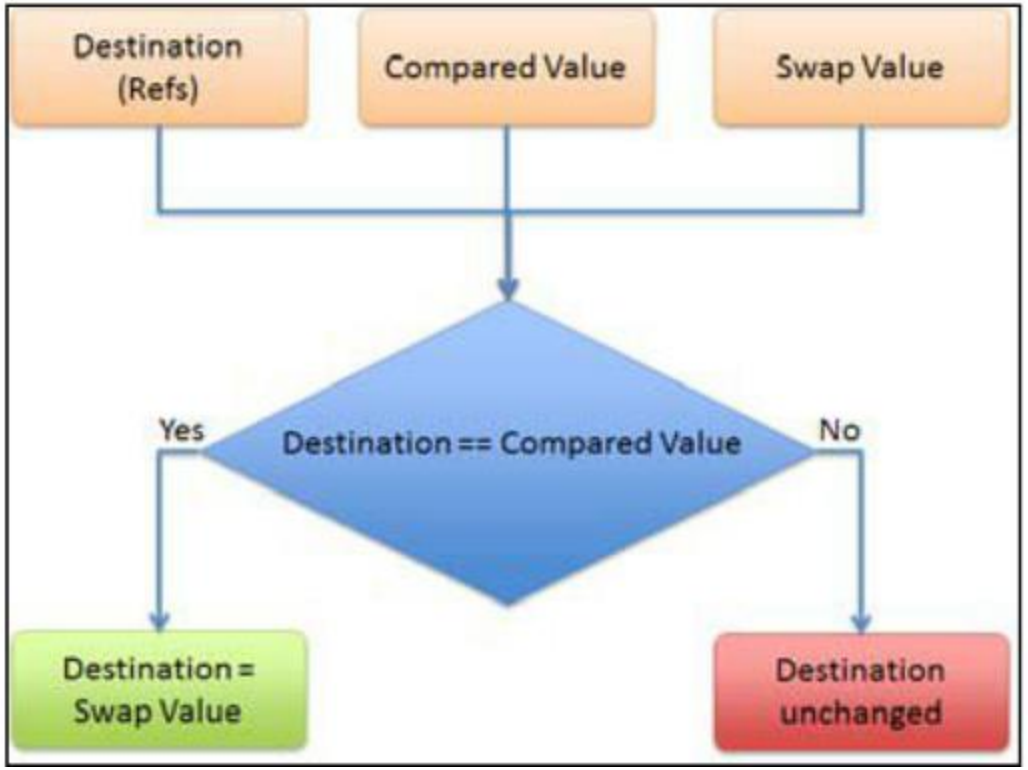
What is Software Transactional Memory ?

Concept of threads is based on the model of synchronized access to shared mutable state.

STM abstract the threading and locking and uses two concepts:

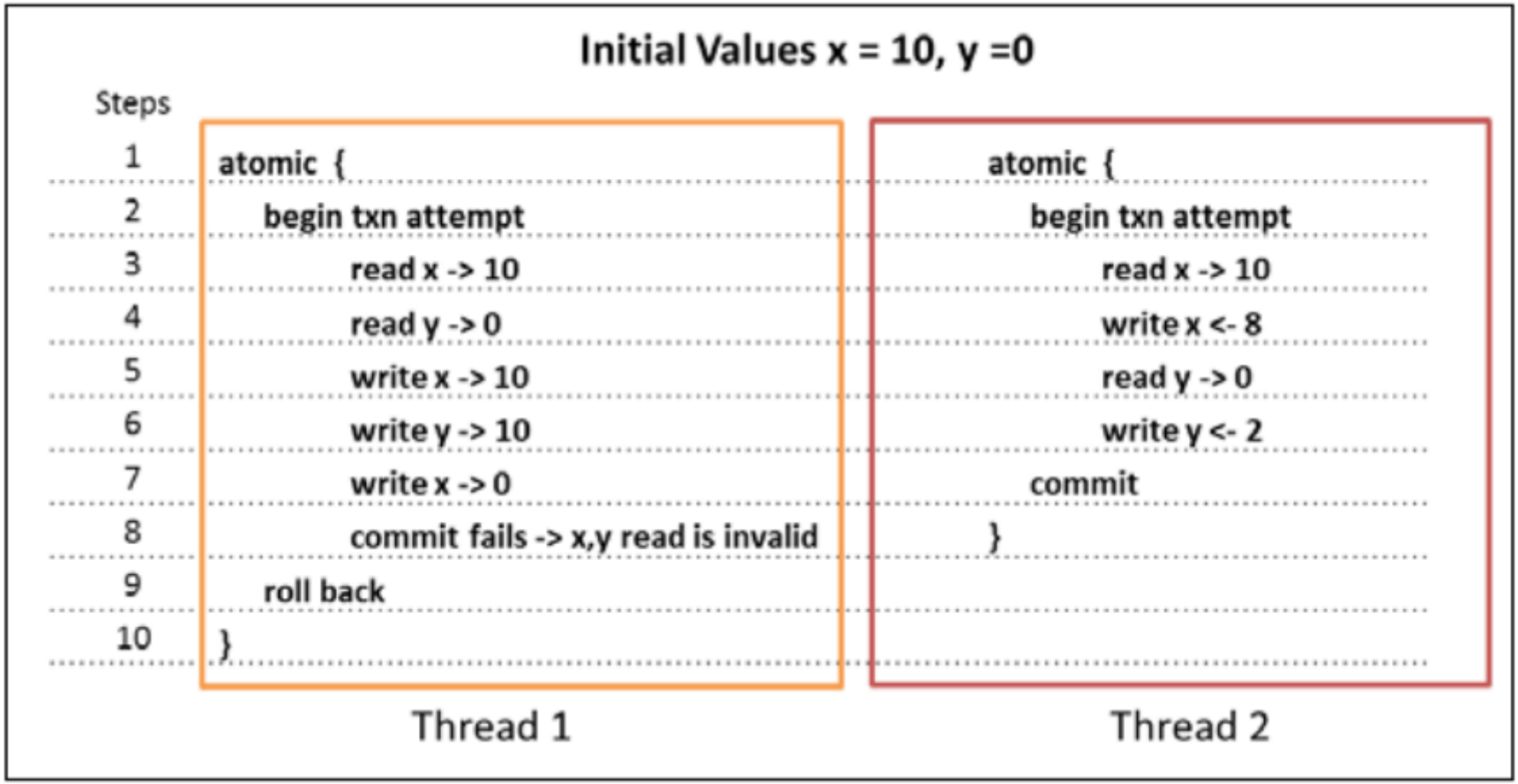
- Optimism - multiples lock a reperformed in parallel,
- Transactions - ACI propreties are preserved

Software Transactional Memory

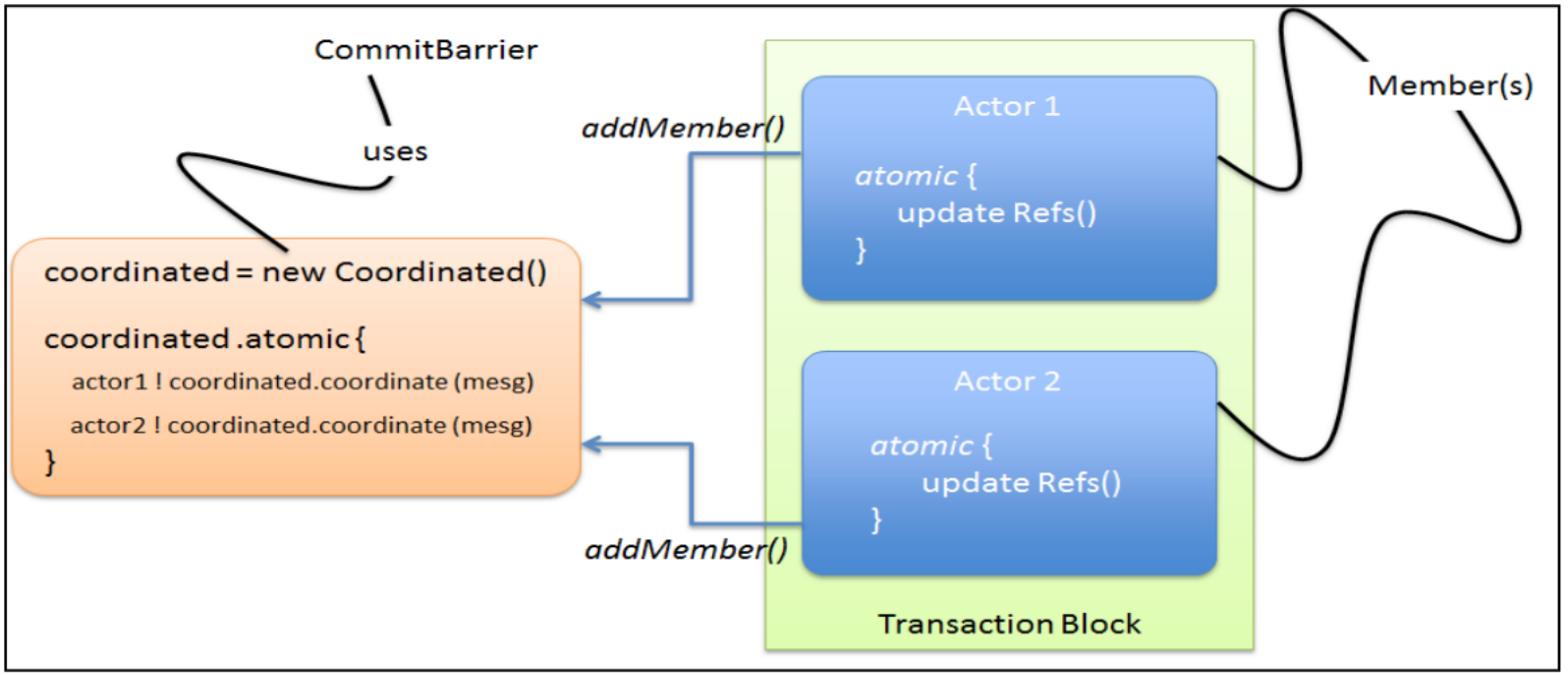


At commit time we compare the values read before transaction at a particular memory location to the value after transaction.

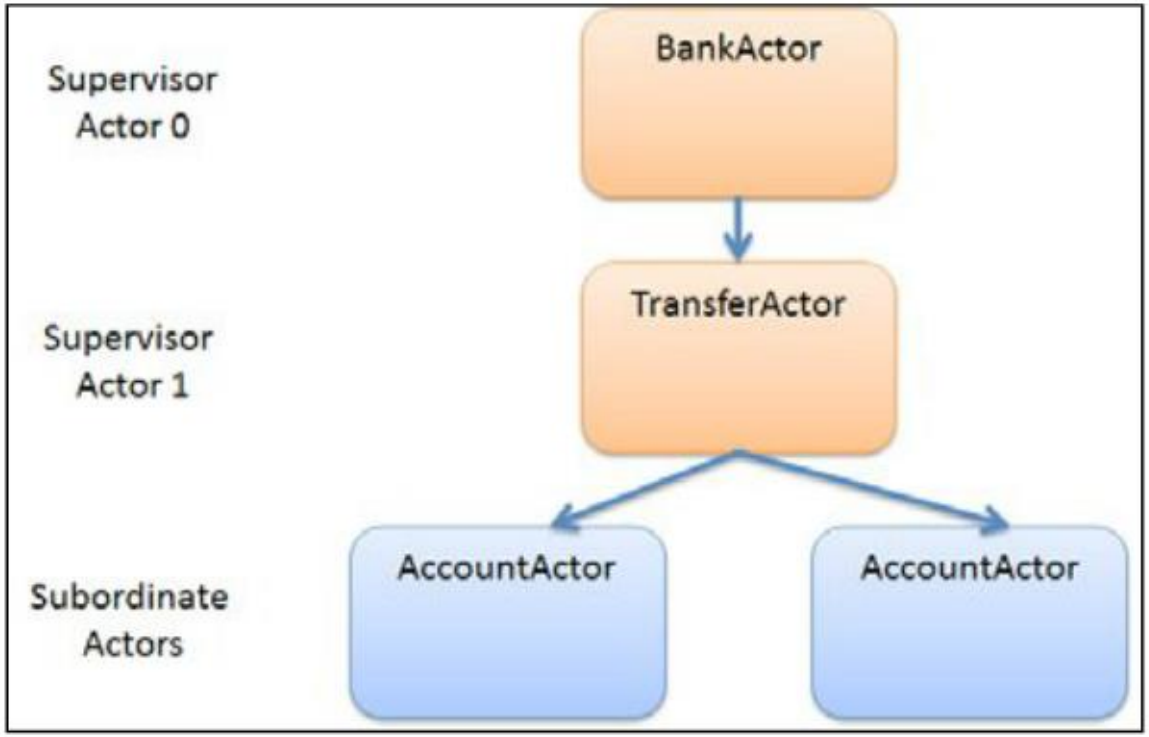
Example


Initial Values x = 10, y =0
Steps
1
2
3
4
5
6
7
8
9
10
atomic {
begin txn attempt
read x -> 10
read y -> 0
write x -> 10
write y -> 10
write x -> 0
commit fails -> x,y read is invalid
roll back
}
atomic {
begin txn attempt
read x -> 10
write x <- 8
read y -> 0
write y <- 2
commit
}
Thread 1
Thread 2

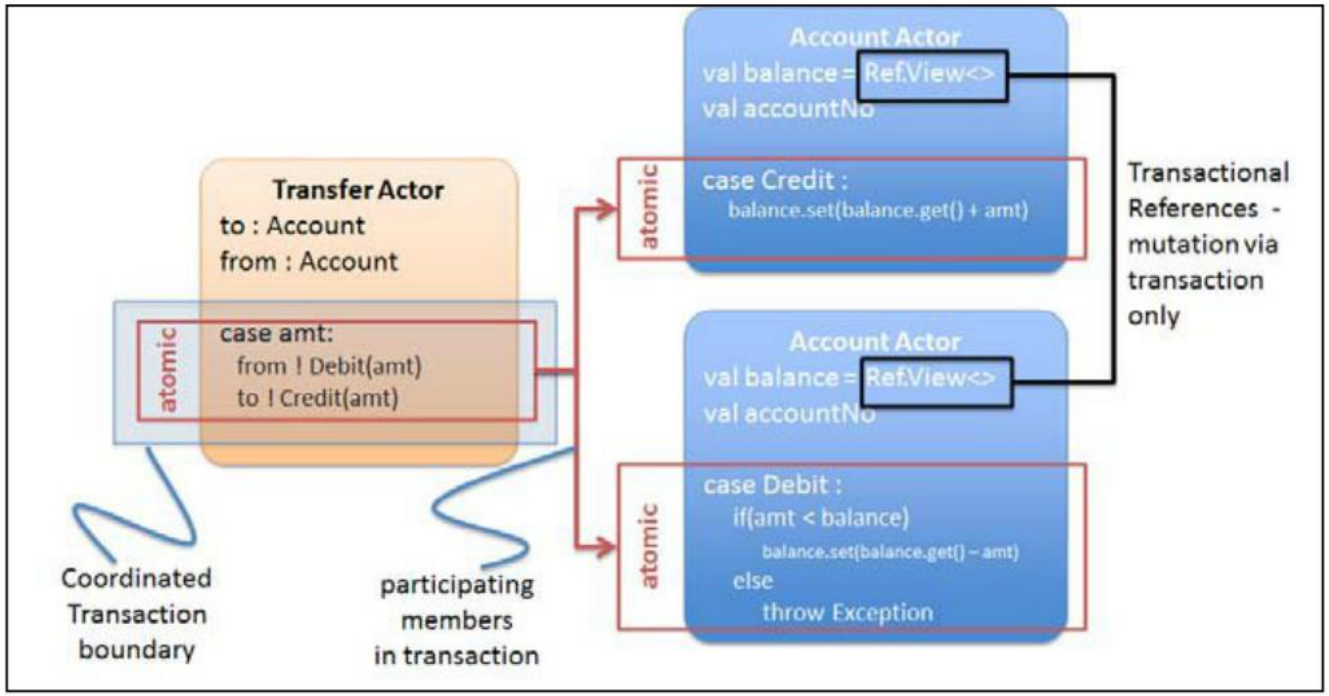
Coordinating Transaction across Actors



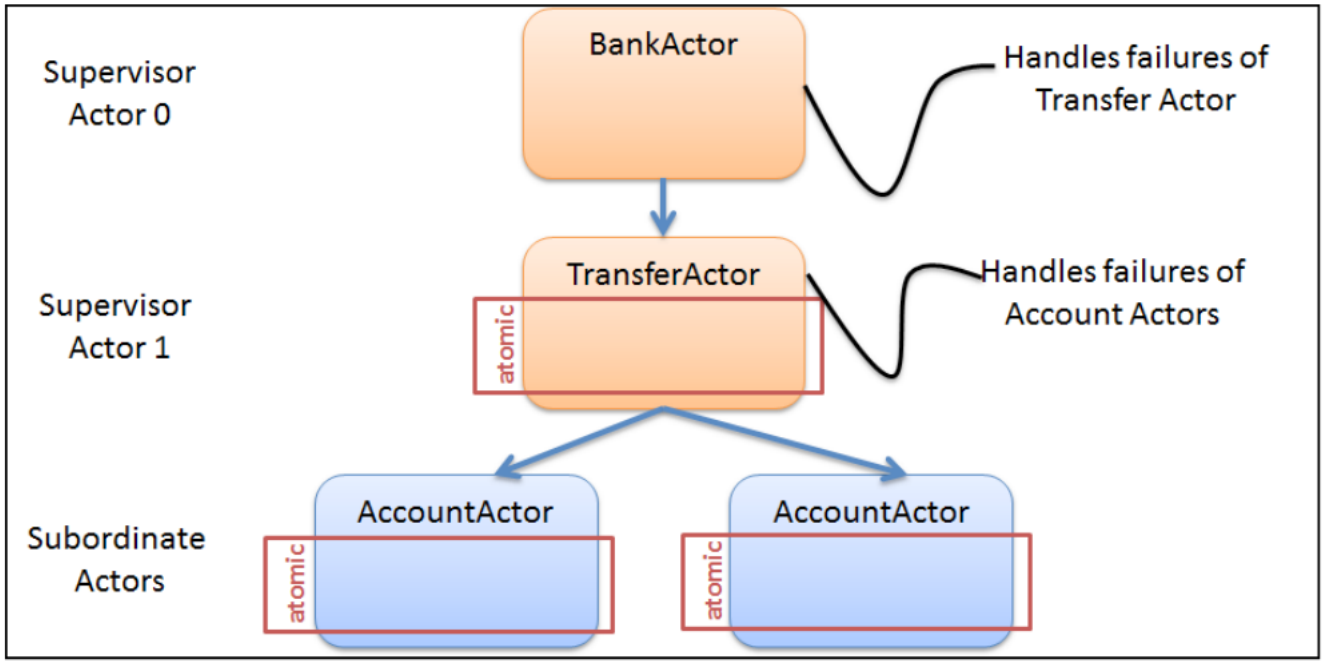
Money Transfer Example



Money Transfer Example



Money Transfer Example



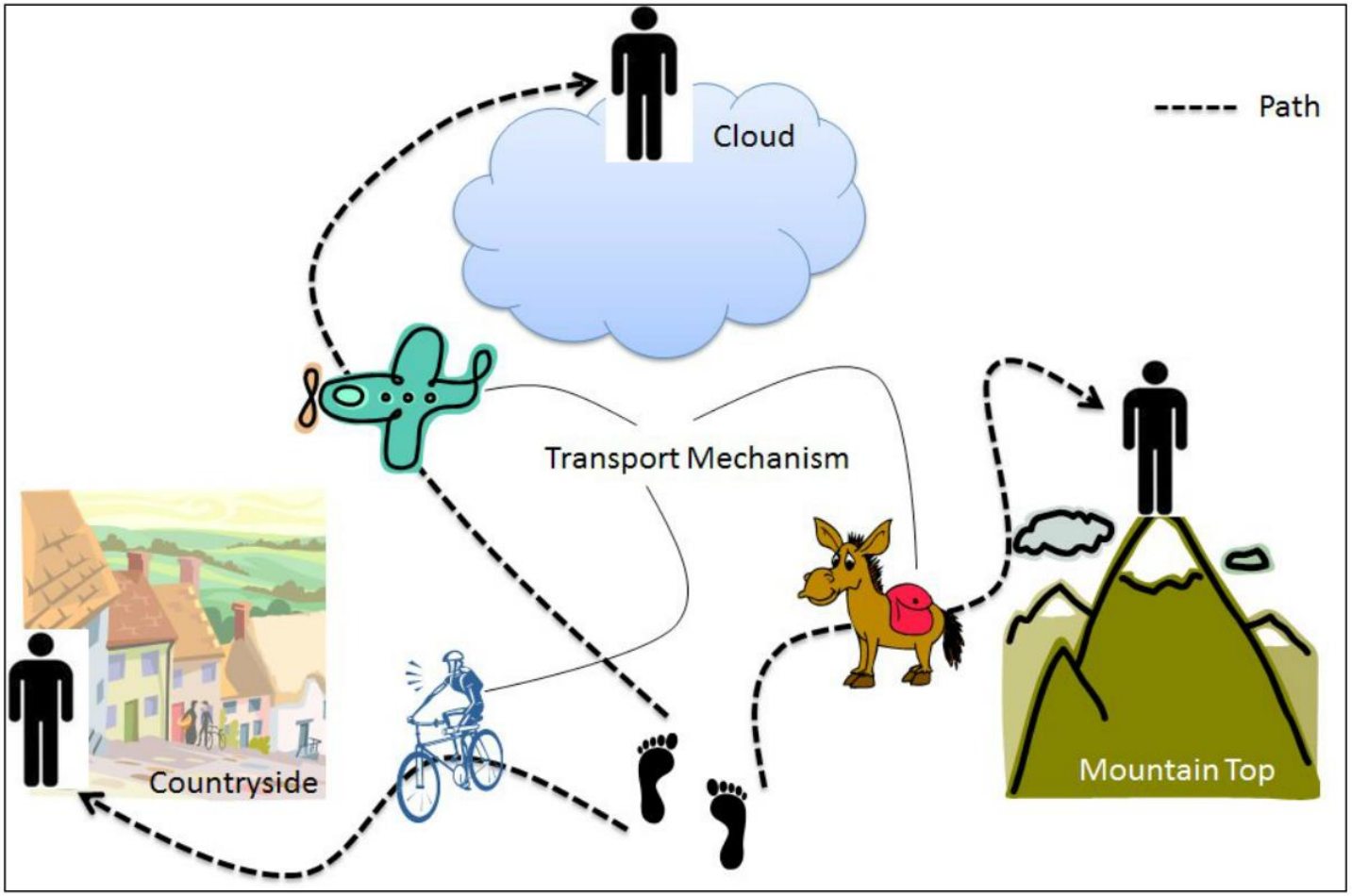
Agents

Agents provide independent asynchronous change of individual locations and shared access to the immutable state.

Agents run on a different threads.

Agents are integrated with STM.

Distributed Computing



Requires ability to locate where actors are running

Actor path

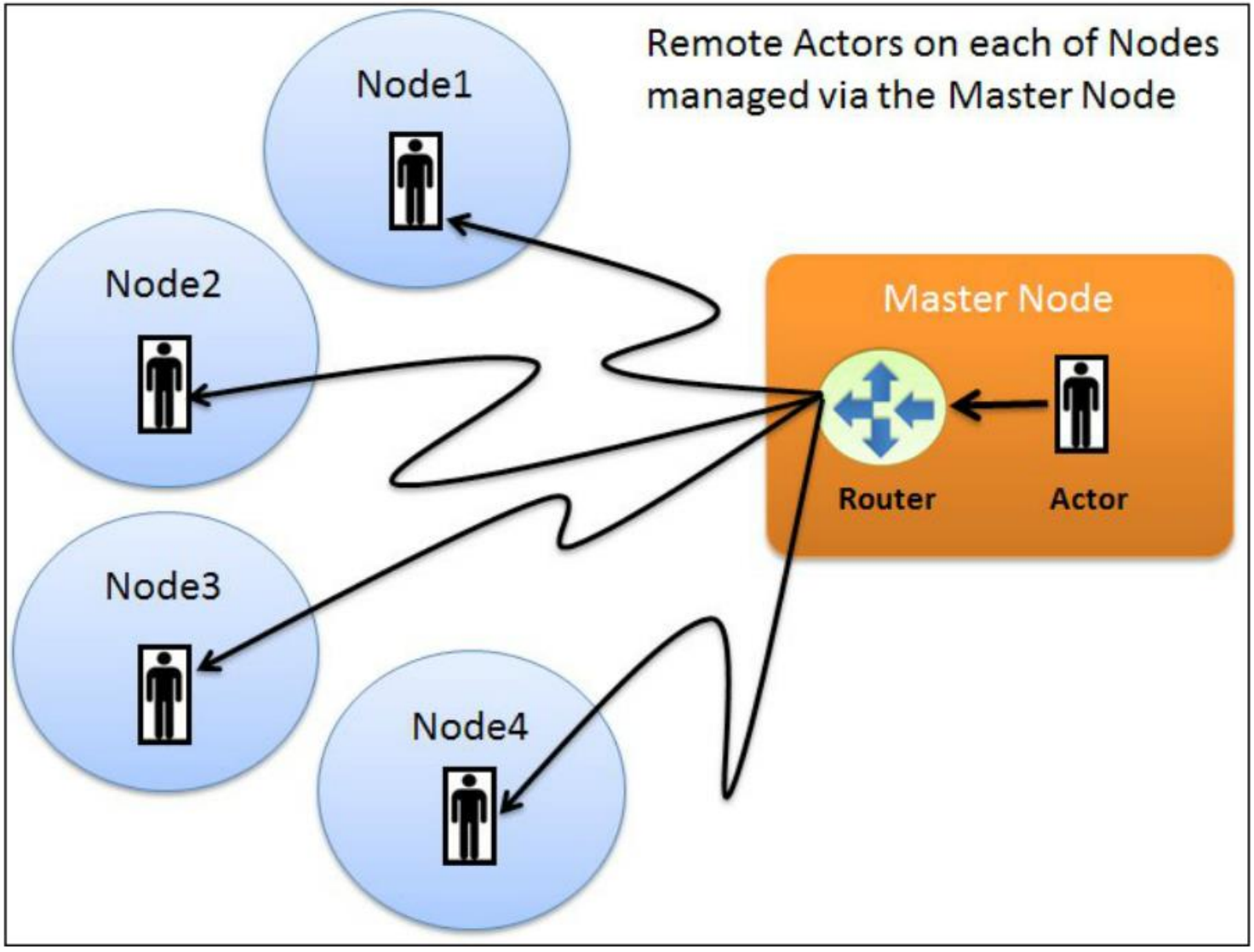
```
akka://<ActorSystem-name>@<hostname>:<port>/<actor path>
```

```
akka://ServerSys@10.102.141.77:2552/user/SomeActor
```

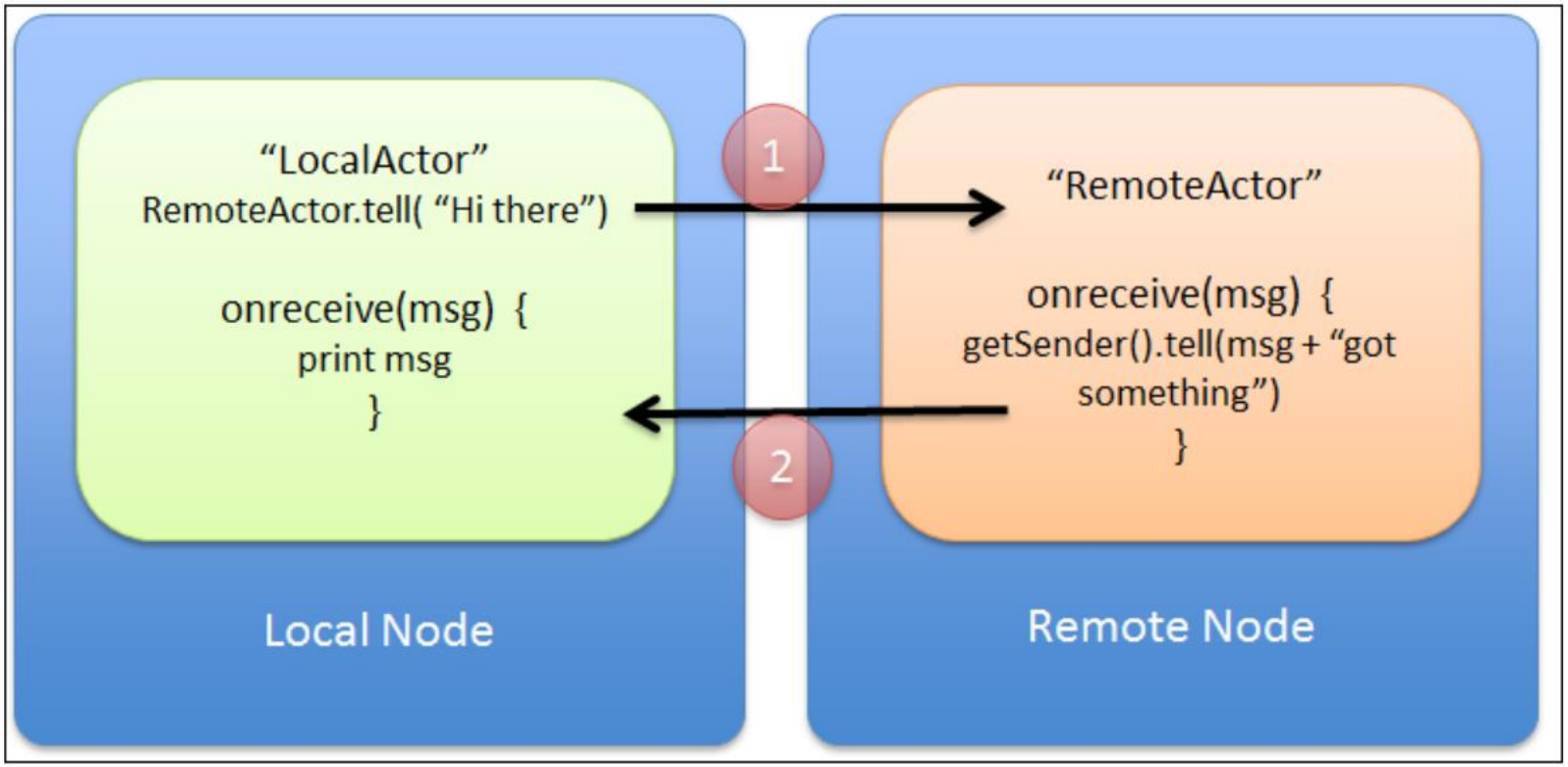
Remote Actors

- Remote actor setup
- Remote actor lookup
- Remote actor deployment
- Routing and data serialization for over-the-wire transmission
- Remote events generated by remote clients and servers, and how we can tap into them

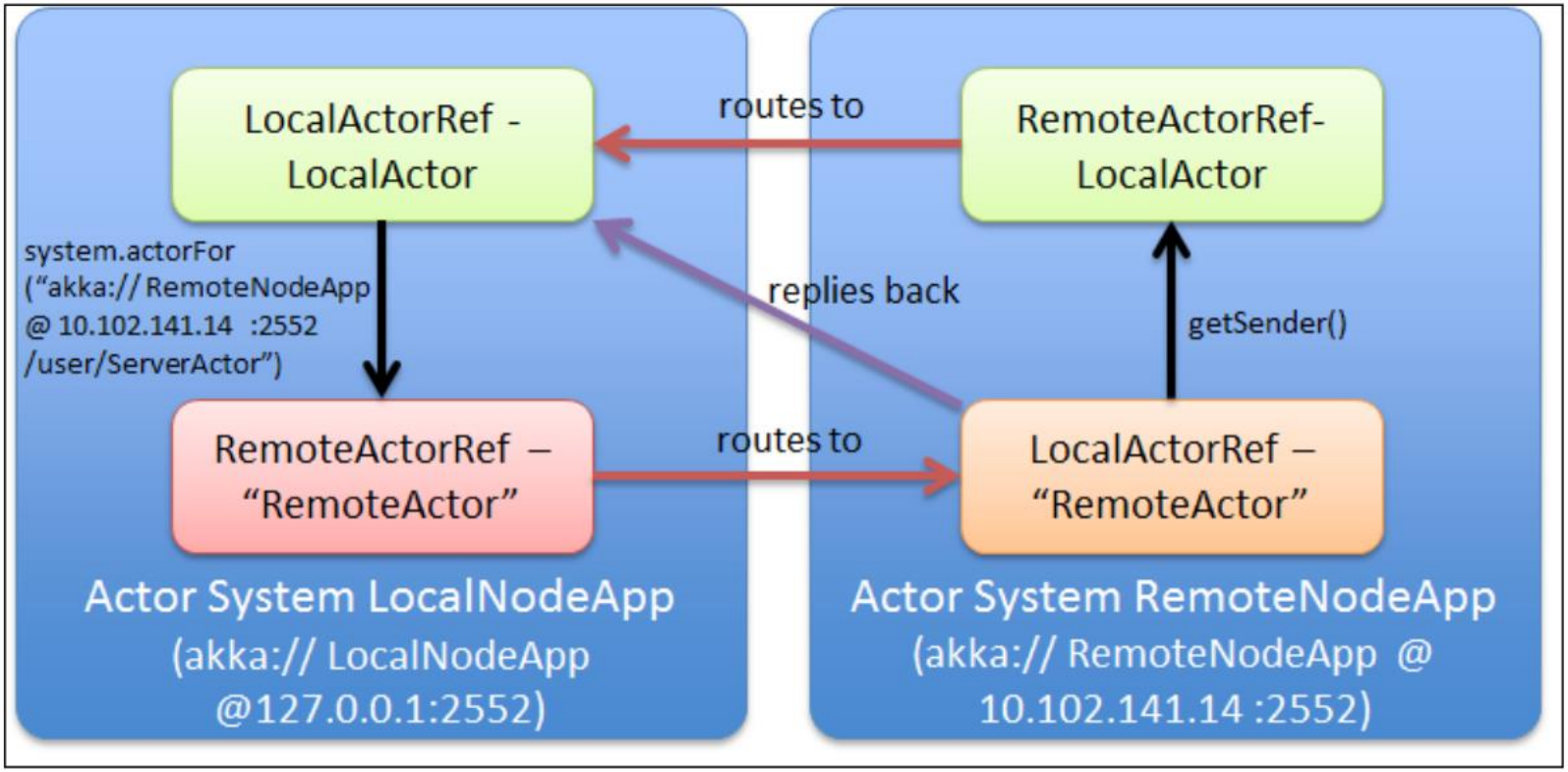
Remote Actors



Remote Actors



Remote Actors



Get reference to Remote Actor

```
//Get a reference to the remote actor
val remoteActor = context.actorFor
    ("akka://RemoteNodeApp@10.102.141.14:2552/user/remoteActor")
```

```
public void onReceive(Object message) throws Exception {
  Future<Object> future = Patterns.ask(remoteActor,
                    message.toString(),timeout);
  String result = (String) Await.result(future,
                      timeout.duration());
  log.info("Message received from Server -> {}", result);
}
```

Creating Remote Actor Programmatically I

Java:

```
Address addr = new Address("akka", "RemoteNodeApp", "10.102.141.14",
2552);

remoteActor = system.actorOf(new Props(RemoteActor.class)
              .withDeploy(new Deploy(new RemoteScope(addr))));
```

Creating Remote Actor Programmatically II

application.conf

```
akka {
  actor {
       deployment {
          /remoteActorAddr {
              remote = "akka://RemoteNodeApp@10.102.141.14:2552"
       }
    }
  }
```

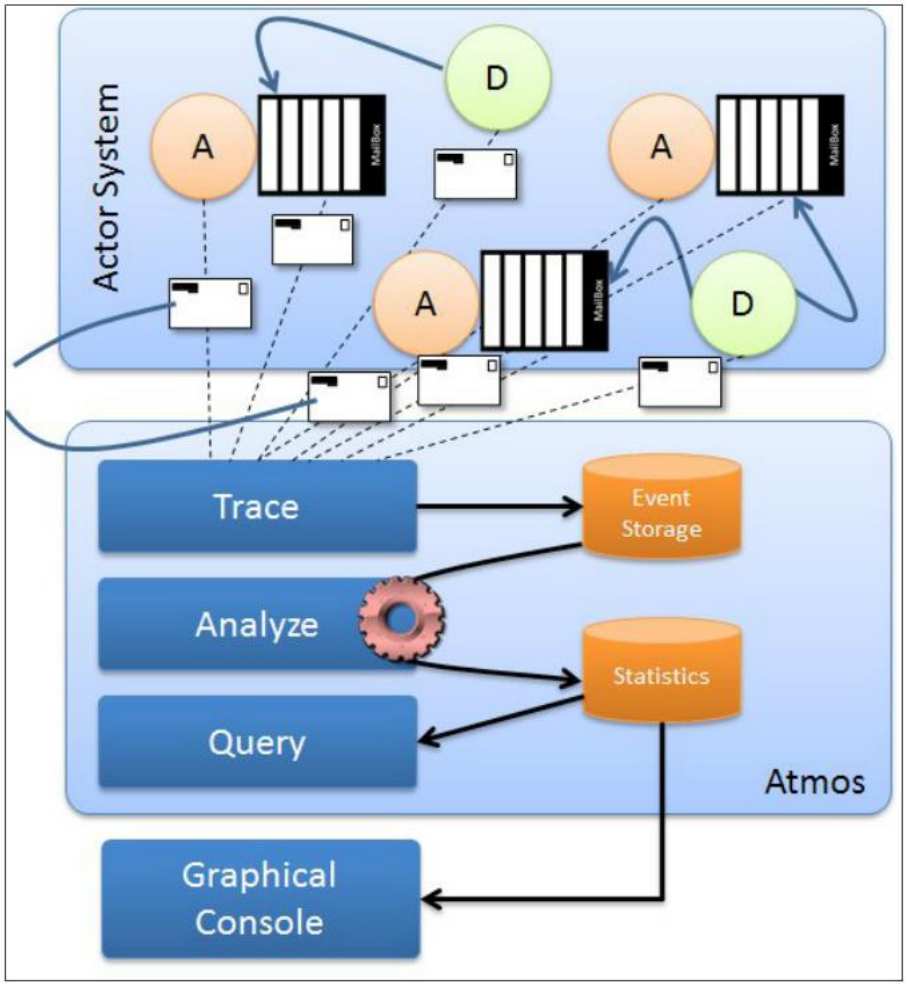
Java:

```
remoteActor = system.actorOf(new Props(RemoteActor.class),
        "remoteActorAddr");
```

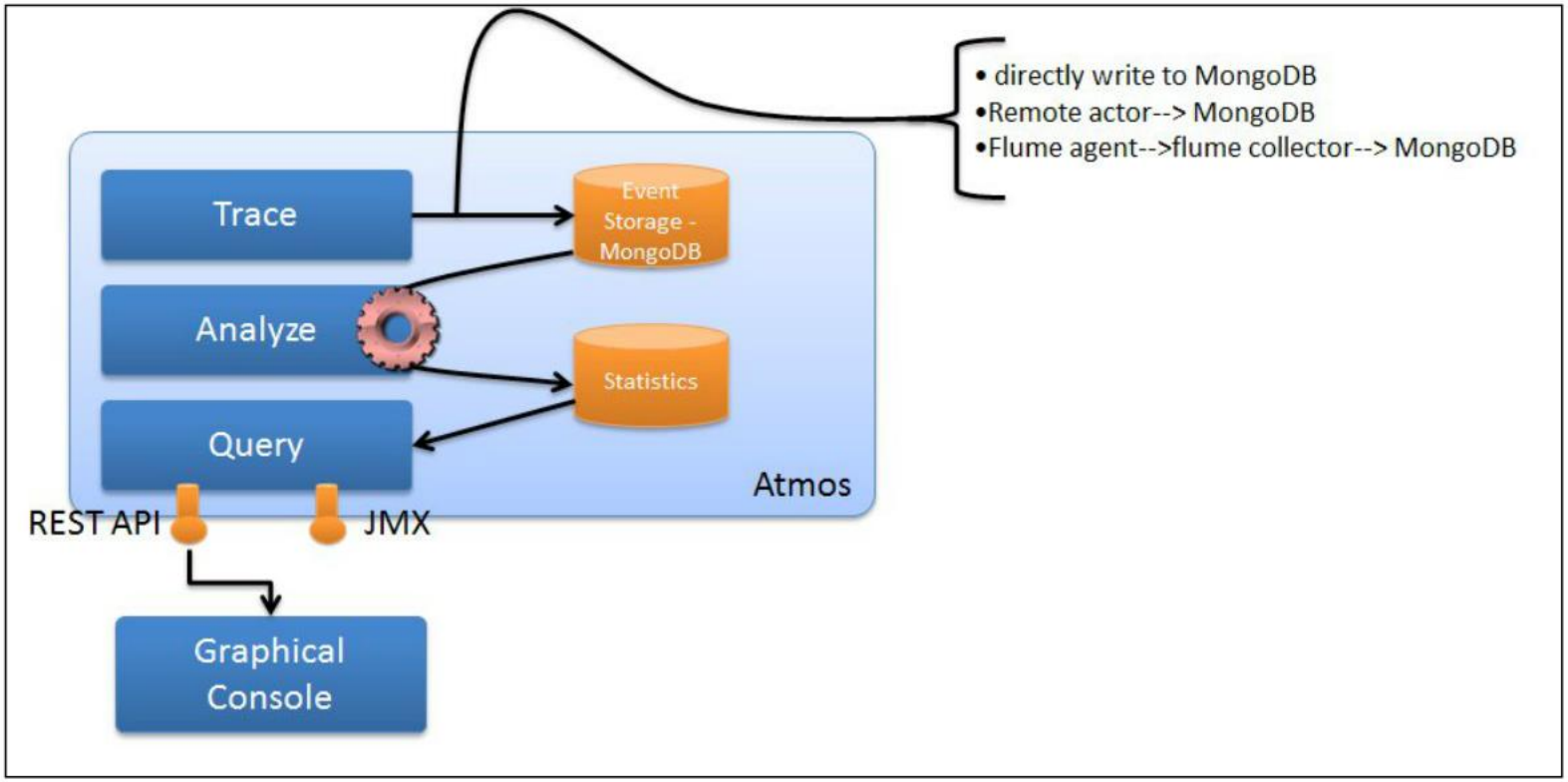
Typesafe Console Modules

- **Trace**: It is responsible for collecting the trace events being emitted by the actor system and storing those trace events in storage.
- **Analyze**: It is responsible for taking raw trace events from storage and producing aggregated results. These results are then stored in a separate storage.
- **Query**: It is responsible for exposing the statistical data via REST and JMX.
- **Typesafe console**: It is the web interface that uses the REST API interface to connect to the query module and provide the rich interface.

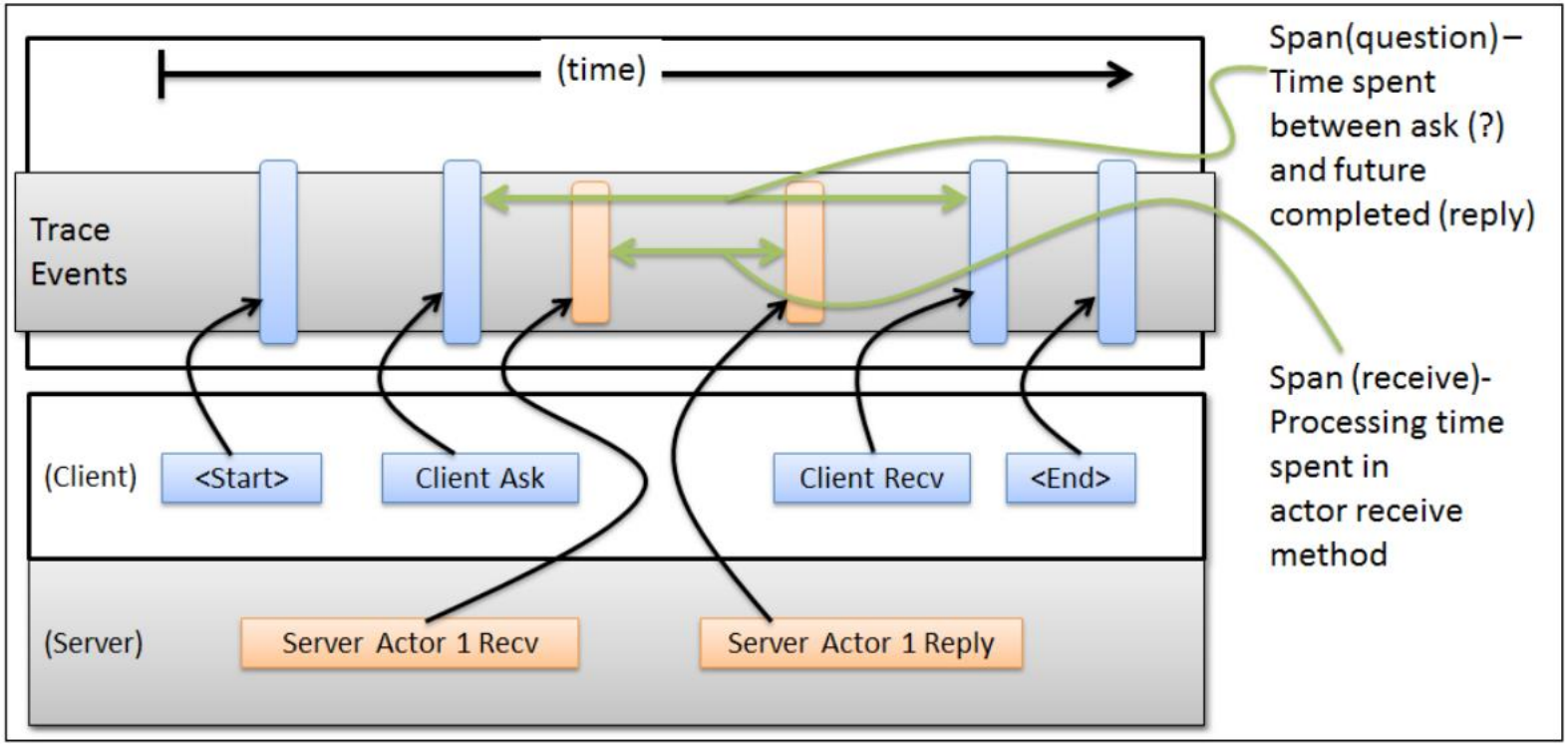
Management


Actor System
A
D
A
A
D
MailBox
Trace
Event Storage
Analyze
Statistics
Query
Atmos
Graphical Console

API


• directly write to MongoDB
•Remote actor--> MongoDB
•Flume agent-->flume collector--> MongoDB
Trace
Event Storage - MongoDB
Analyze
Statistics
Query
Atmos
REST API
JMX
Graphical Console

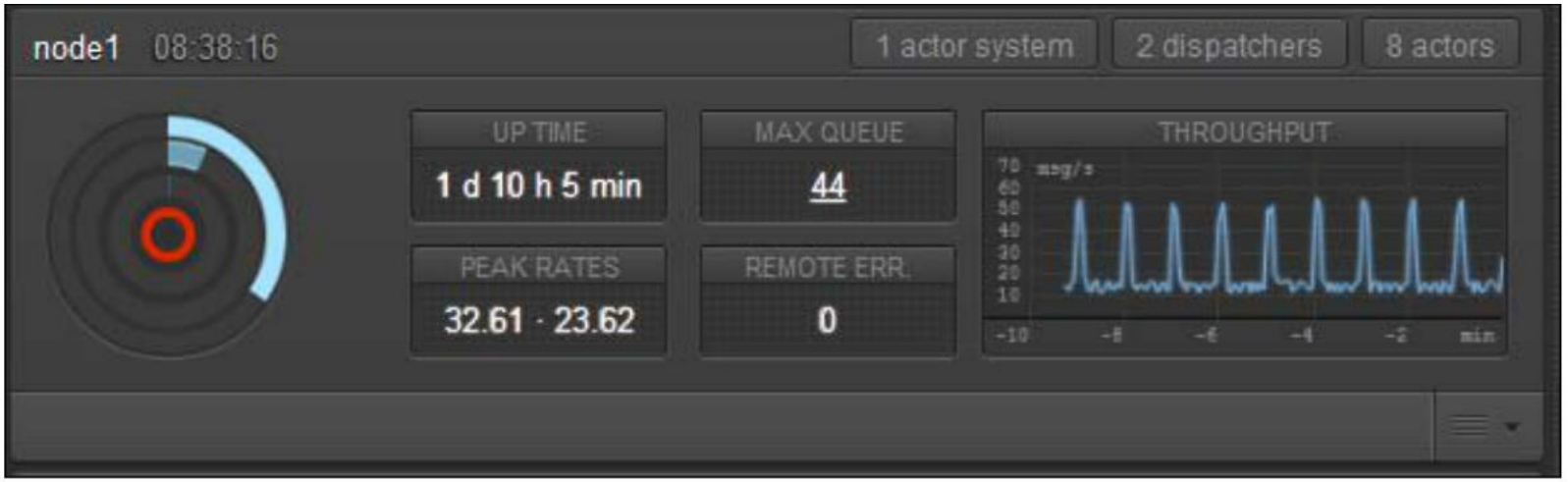
Analyse

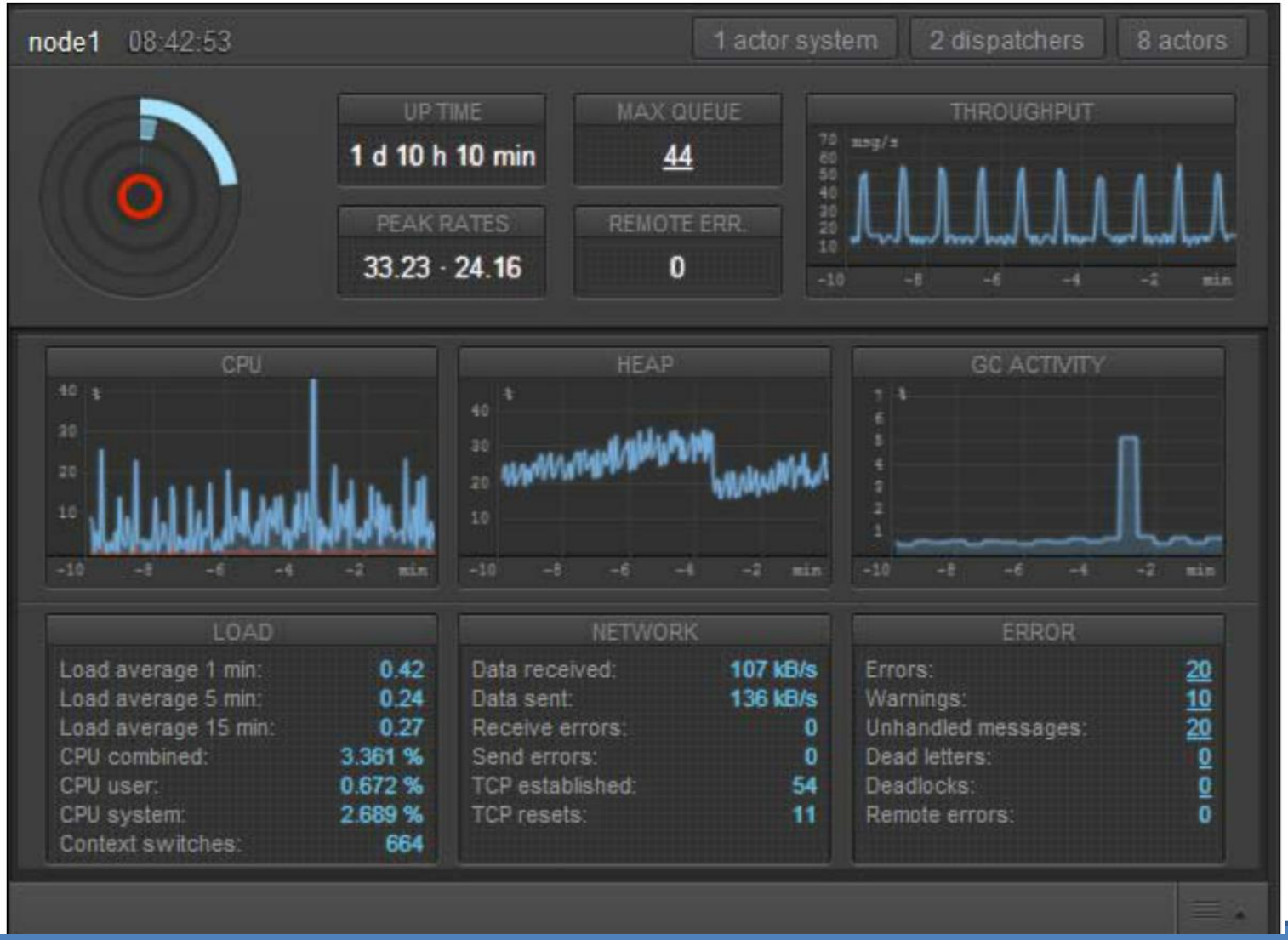

(time)
Trace Events
(Client)
<Start>
Client Ask
Client Recv
<End>
(Server)
Server Actor 1 Recv
Server Actor 1 Reply
Span(question)– Time spent between ask (?) and future completed (reply)
Span (receive)- Processing time spent in actor receive method

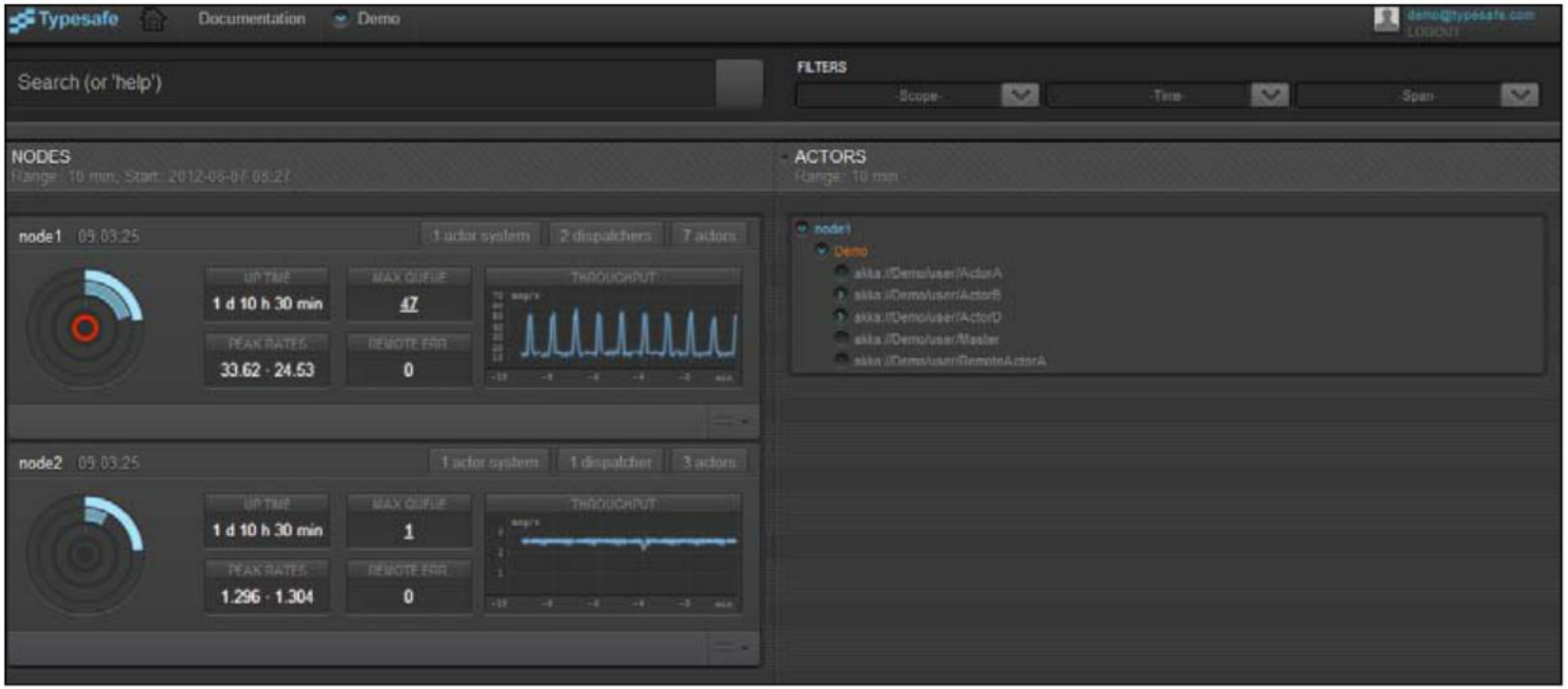
Atmos Supportred Span

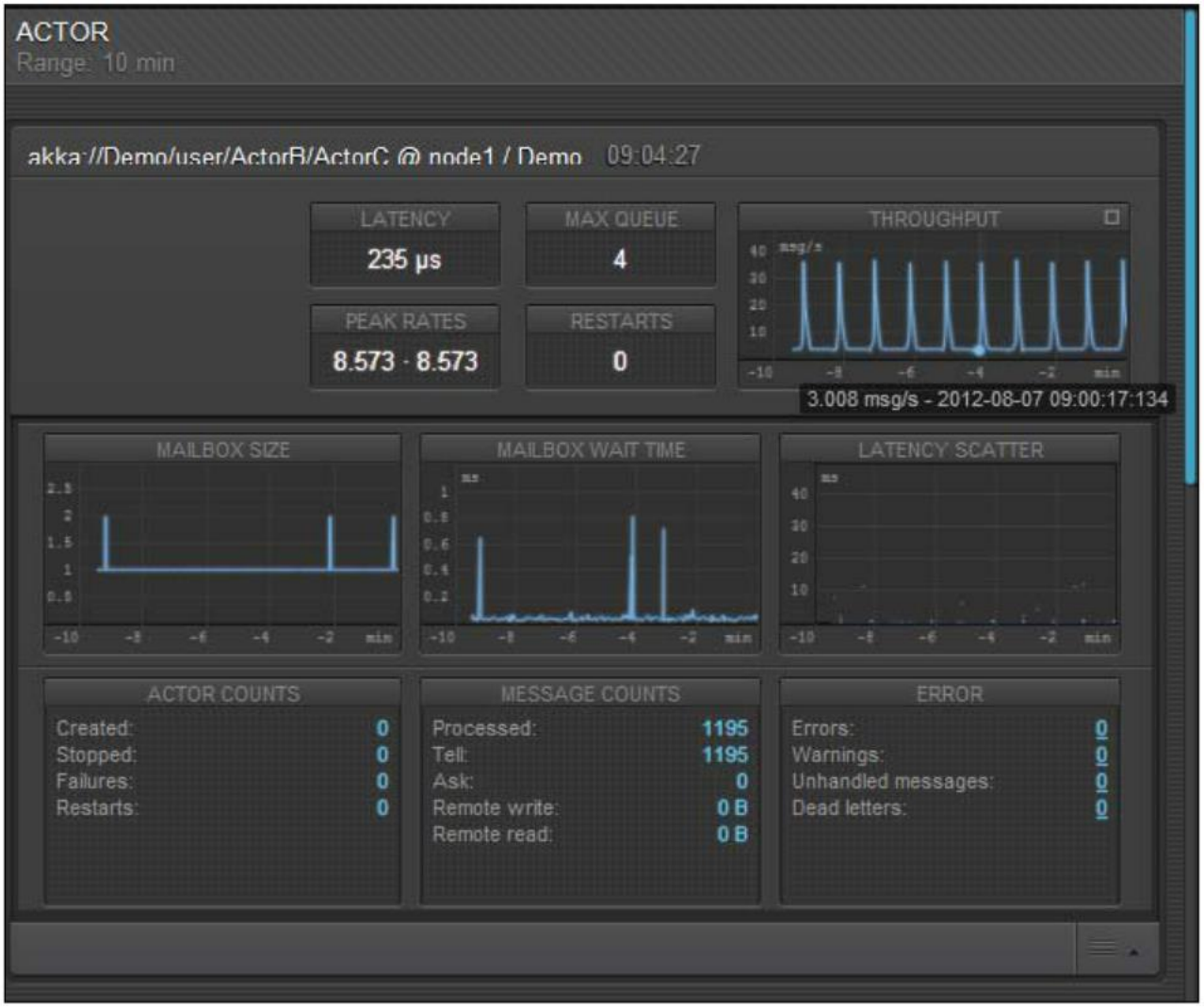
- **Message**: A span for a message, from when it is sent until the processing of the message is completed
- **Mailbox**: A span for the waiting time of the message in the actor mailbox
- **Receive**: A span for the processing time in the actor receive method
- **Question**: A span between ask (?) and future completed (reply)
- **Remote**: A span for remote latency
- **Any user-defined marker span**: A user-defined span may start and end at any location in the message trace; that is, it may span over several actors

Graphical Console


node1
08:38:16
1 actor system
2 dispatchers
8 actors
UP TIME
1 d 10 h 5 min
MAX QUEUE
44
PEAK RATES
32.61 · 23.62
REMOTE ERR.
0
THROUGHPUT
msg/s
70
60
50
40
30
20
10
-10
-8
-6
-4
-2
min


node1 08:42:53
1 actor system
2 dispatchers
8 actors
UP TIME
1 d 10 h 10 min
MAX QUEUE
44
PEAK RATES
33.23 · 24.16
REMOTE ERR.
0
THROUGHPUT
msg/s
CPU
HEAP
GC ACTIVITY
LOAD
Load average 1 min: 0.42
Load average 5 min: 0.24
Load average 15 min: 0.27
CPU combined: 3.361 %
CPU user: 0.672 %
CPU system: 2.689 %
Context switches: 664
NETWORK
Data received: 107 kB/s
Data sent: 136 kB/s
Receive errors: 0
Send errors: 0
TCP established: 54
TCP resets: 11
ERROR
Errors: 20
Warnings: 10
Unhandled messages: 20
Dead letters: 0
Deadlocks: 0
Remote errors: 0

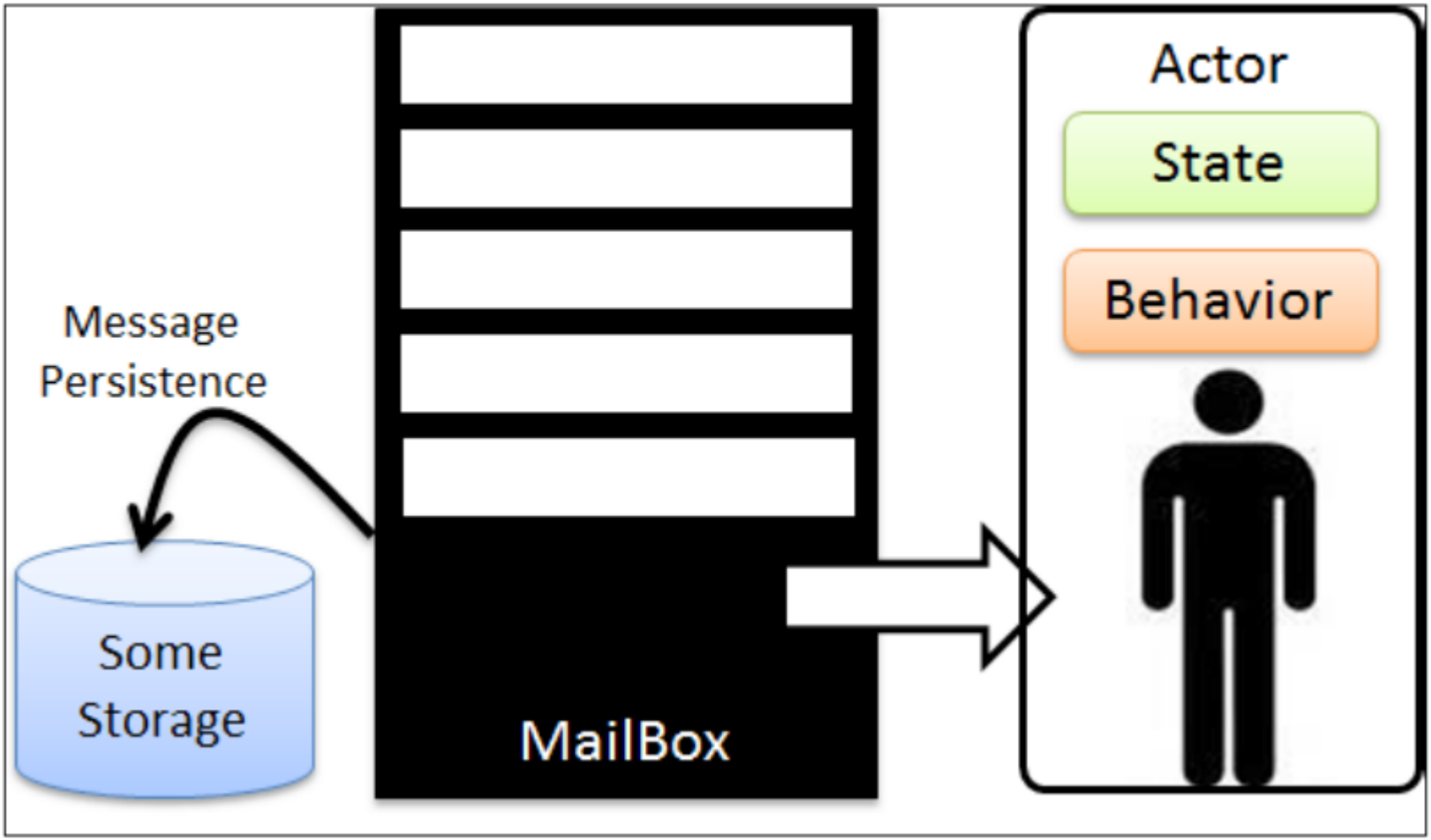

Typesafe
Documentation
Demo
demo@typesafe.com
LOGOUT
Search (or 'help')
FILTERS
Scope
Time
Span
NODES
ACTORS
node1
1 actor system
2 dispatchers
7 actors
UP TIME
1 d 10 h 30 min
MAX QUEUE
47
THROUGHPUT
PEAK RATES
33.62 - 24.53
REMOTE RQR
0
node2
1 actor system
1 dispatcher
3 actors
UP TIME
1 d 10 h 30 min
MAX QUEUE
1
THROUGHPUT
PEAK RATES
1.296 - 1.304
REMOTE RQR
0
node1
Demo
akka://Demo/user/ActorA
akka://Demo/user/ActorB
akka://Demo/user/ActorD
akka://Demo/user/Master
akka://Demo/user/RemoteActorA


ACTOR
Range: 10 min
akka://Demo/user/ActorB/ActorC @ node1 / Demo 09:04:27
LATENCY
235 µs
MAX QUEUE
4
PEAK RATES
8.573 · 8.573
RESTARTS
0
THROUGHPUT
msg/s
3.008 msg/s - 2012-08-07 09:00:17:134
MAILBOX SIZE
MAILBOX WAIT TIME
ms
LATENCY SCATTER
ms
ACTOR COUNTS
Created: 0
Stopped: 0
Failures: 0
Restarts: 0
MESSAGE COUNTS
Processed: 1195
Tell: 1195
Ask: 0
Remote write: 0 B
Remote read: 0 B
ERROR
Errors: 0
Warnings: 0
Unhandled messages: 0
Dead letters: 0

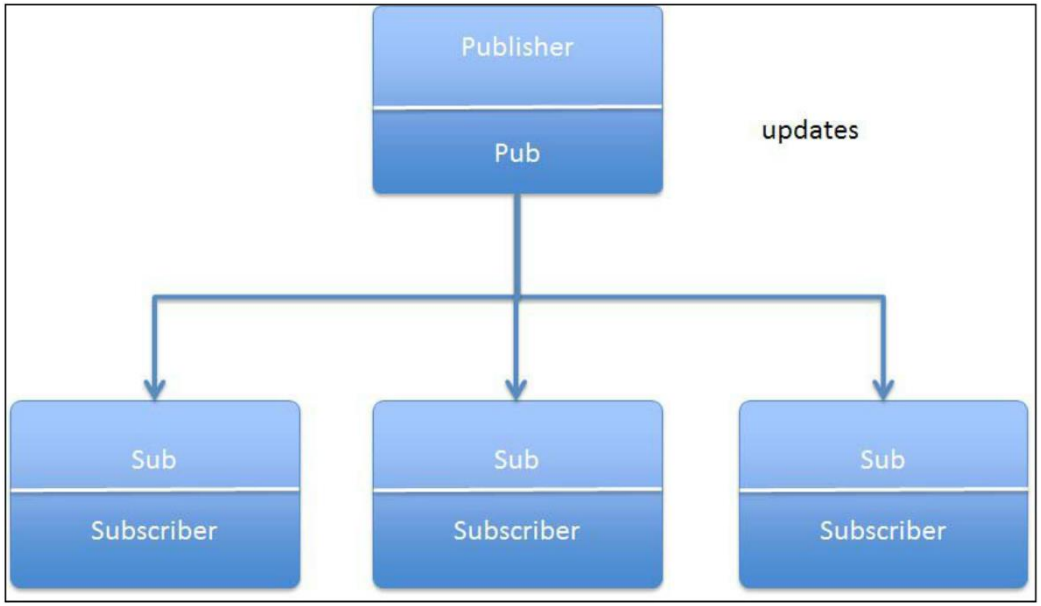
Advance Topics

- Durable mailboxes
- Actors and web applications
- Integrating actors with ZeroMQ

Durable Mailbox



Integrating AKKA with ZeroMQ



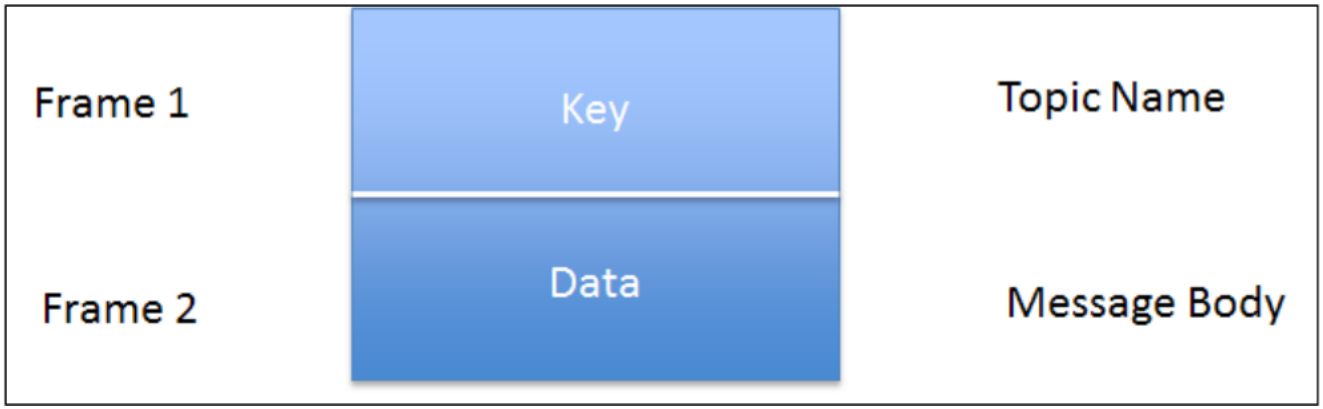
Akka supports the following connectivity patterns using ZeroMQ:

- Publisher-subscriber connection
- Request-reply connection
- Router-dealer connection
- Push-pull connection

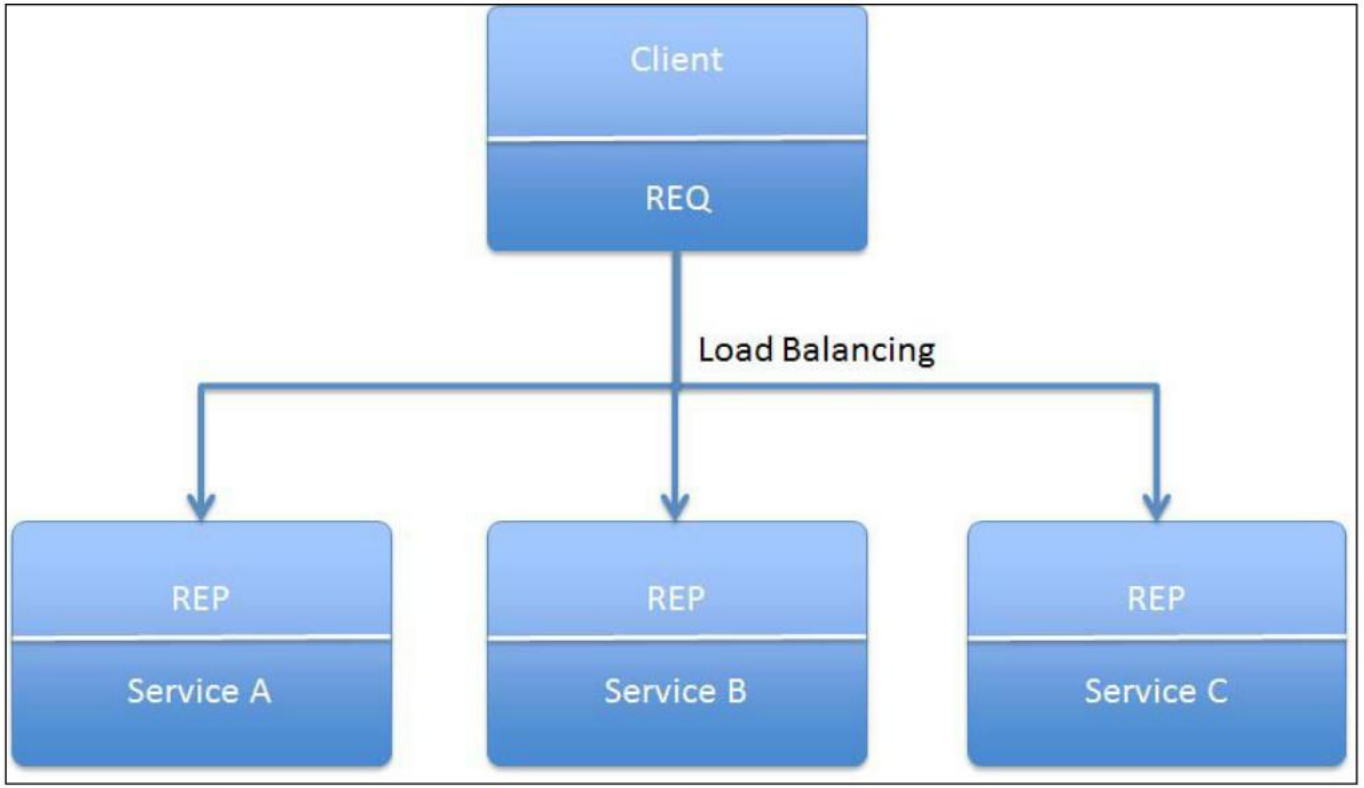
Durable Mailbox Support

- **File**: Using a transaction log on a local filesystem
- **Redis**: Using the open source, key-value store (`http://www.redis.io/`)
- **Zookeeper**: A centralized service for maintaining configuration information, and naming, providing distributed synchronization, and group services (`http://zookeeper.apache.org/`)
- **Beanstalkd**: Using the work queue feature coupled with `memcache` (`http://kr.github.com/beanstalkd/`)
- **MongoDB**: Using the open source NoSQL database storage (`http://www.mongodb.org/`)

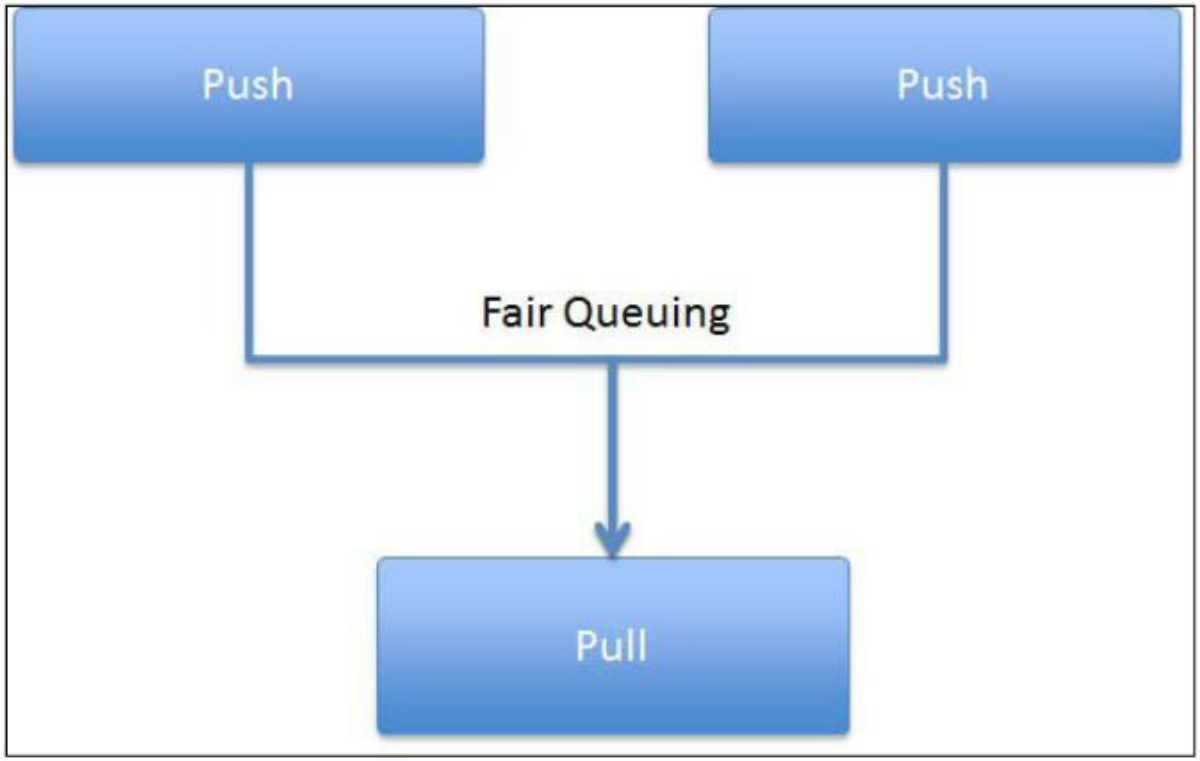
Message in ZeroMQ



Request Reply Connection



Non-blocking Request/Reply



END

http://download.csdn.net/detail/u012825909/7991901