Concurrent access to shared mutable objects in Encore and its GC related implications

2015-09-28

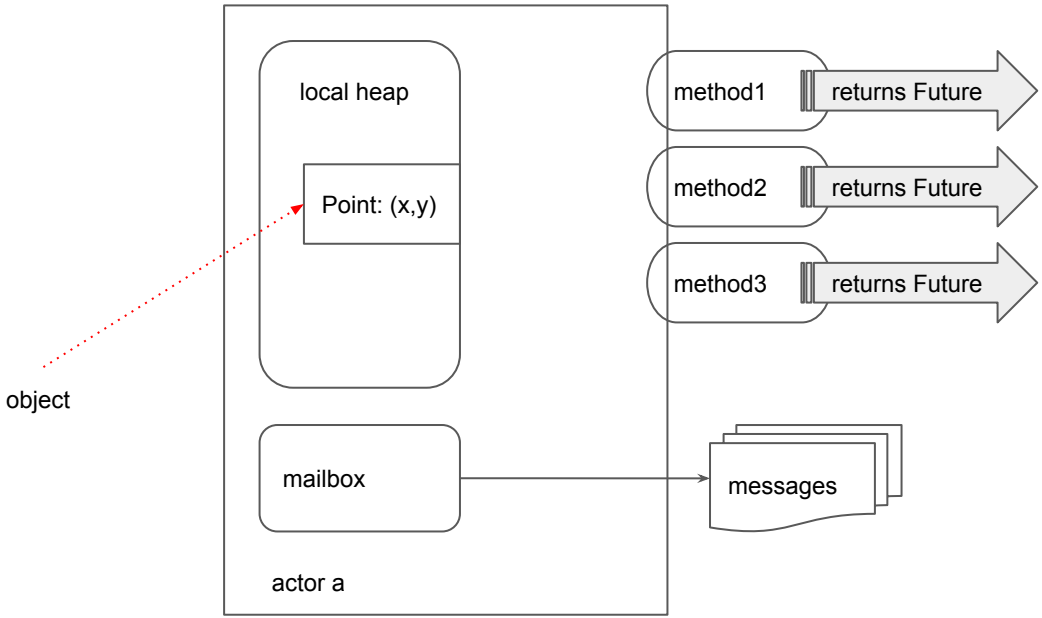
Tobias Wrigstad && **Albert Yang**
albert.yang@it.uu.se

Outline

- Encore Programming Model
- Sharing mutable objects
- How Object GC works in Encore
- Example of Object GC failure
- How Object GC in other actor-based systems work
- Shared Object
- Summary

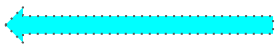
Encore Programming Model

- Active objects (actor)
  - own thread of control
  - async interface
  - local heap
  - mailbox
- Passive objects (object)
  - like plain old Java object
  - sync interface
  - reside in local heap of actors



Sharing mutable objects

- Using active objects
  - high latency if many actors scheduled in front
  - memory overhead; each object carries a local heap
  - serial access even for non-overlapping operations
- Using passive objects
  - Data Race
    i. mutates objects after sending
    ii. mutates objects after receiving



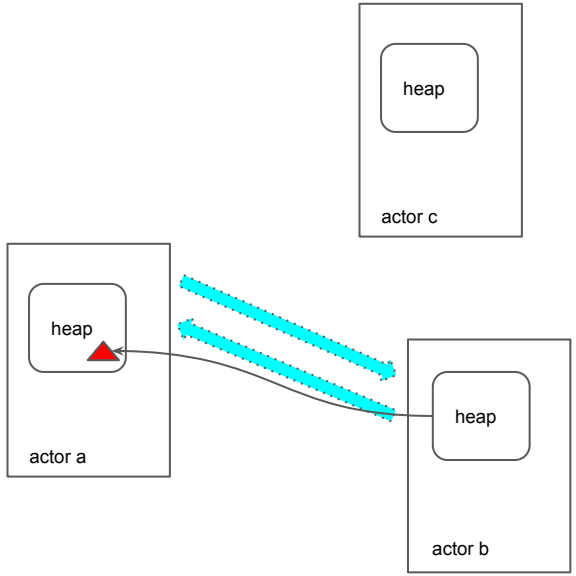
solvable using lock, STM, type system...

  - Breaks parallel GC
    i. parallel GC on local heap of each actor
    ii. PonyRT deals with sharing immutable objects
    iii. but sharing mutable objects would lead to dangling pointers

GC in Encore

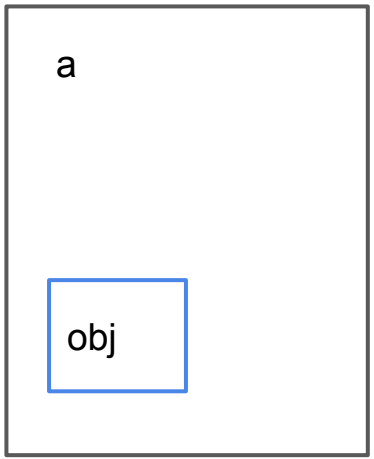
PonyRT inside!

- Actor GC [Clebsch, Drossopoulou ‘13]
  - detect unreachable actors and collect them
- Object GC [Clebsch et_al. ‘15]
  - based on ownership and reference counting
  - local GC on the heap to each actor
  - sync using message passing



actor **c** could GC independently, while actor **a** and **b** syncs using message passing

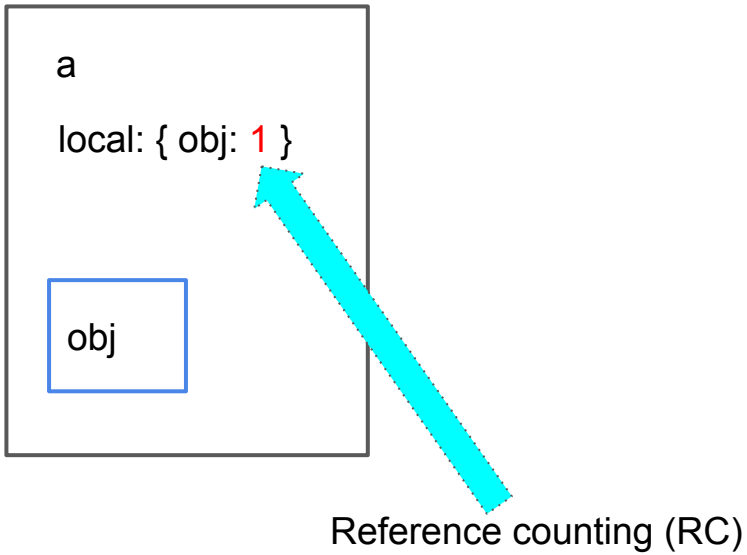
Object GC (1)




```
class A {
  b = new B -- another actor
  obj = new Obj
  b.method(obj)
}

class B {
  def method(o : Obj) {
    print “recv”
  }
}
```

actor **a** creates **obj** in its local heap

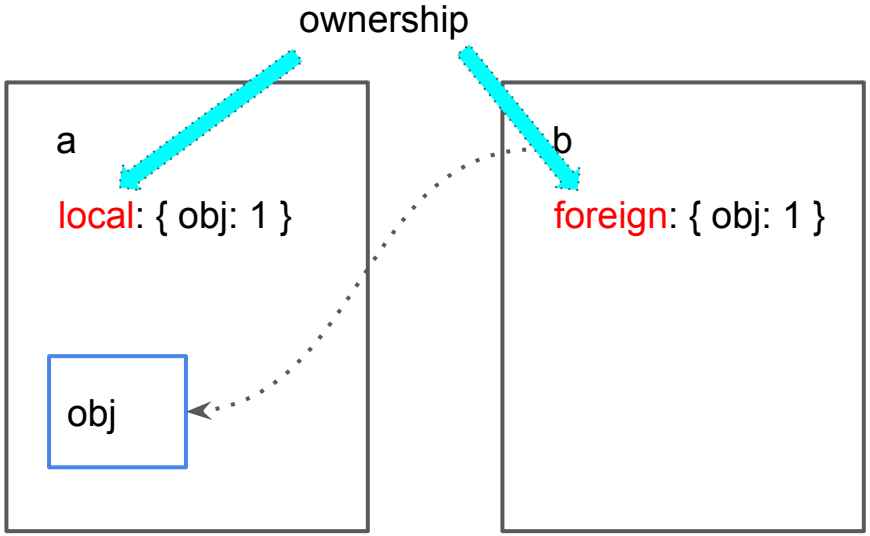
Object GC (2)




```
class A {
  b = new B -- another actor
  obj = new Obj
  b.method(obj)
}

class B {
  def method(o : Obj) {
    print “recv”
  }
}
```

actor **a** creates an entry in **`local`** set before sending it out

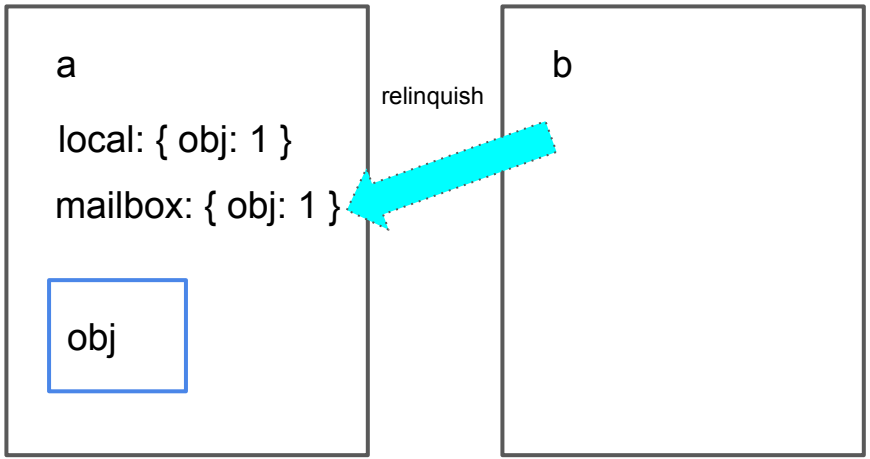
Object GC (3)




```
class A {
  b = new B -- another actor
  obj = new Obj
  b.method(obj)
}

class B {
  def method(o : Obj) {
    print “recv”
  }
}
```

actor **b** receives **obj** and creates an entry in **foreign** set, for it doesn’t belong to the current actor

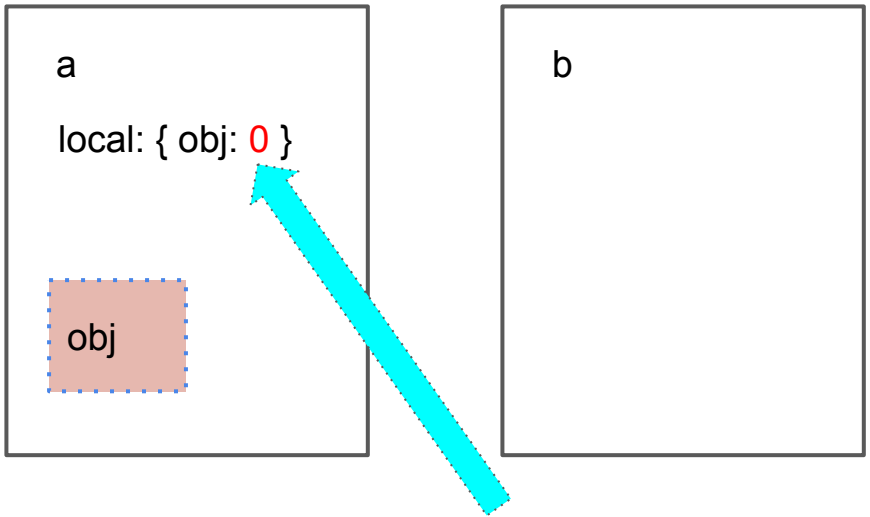
Object GC (4)




```
class A {
  b = new B -- another actor
  obj = new Obj
  b.method(obj)
}

class B {
  def method(o : Obj) {
    print “recv”
  }
}
```

During GC in **b**, **obj** is found unreachable and reports to the owner, **a**

Object GC (5)

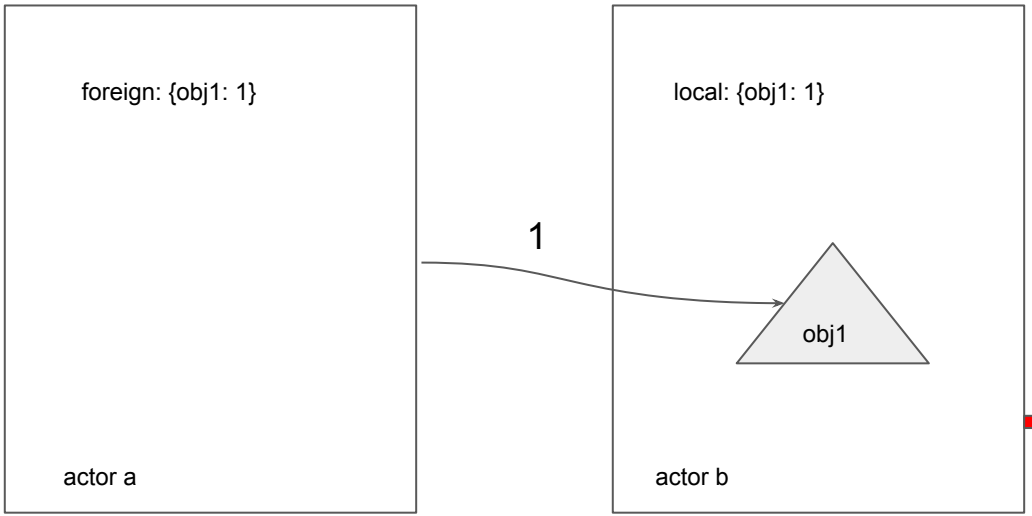



```
class A {
  b = new B -- another actor
  obj = new Obj -- object in local heap
  b.method(obj)
}

class B {
  def method(o : Obj) {
    print “recv”
  }
}
```

RC reaches zero, eligible for collecting

In the next GC cycle, **`obj`** would be collected

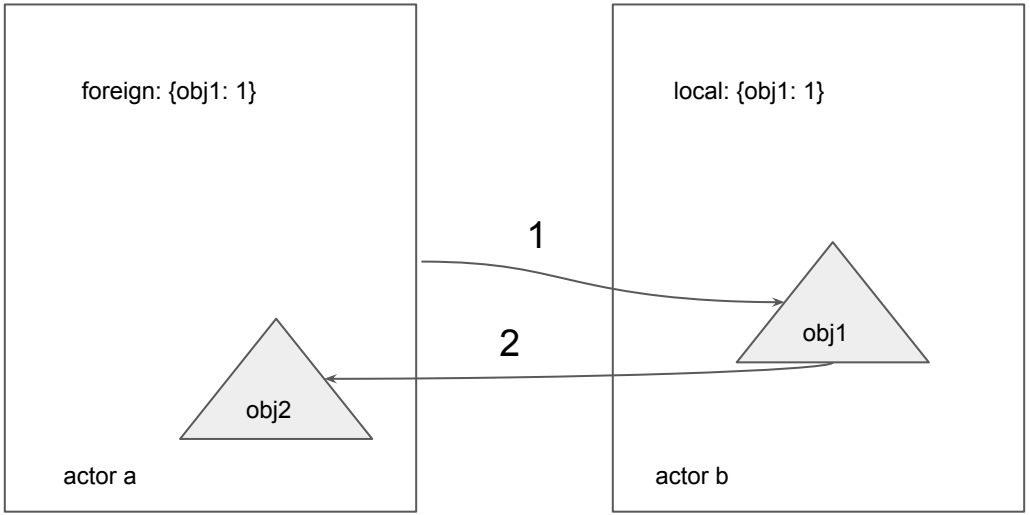
Example of extending objects after receiving (1)




```
class A {
  def method(o : Obj) {
    o.setNext(new Obj)
    o
  }
}

class B {
  a = new A
  obj = new Obj
  obj = get a.method(obj)
  obj.getNext()
}
```

actor **b** creates **obj1** and shares with actor **a**

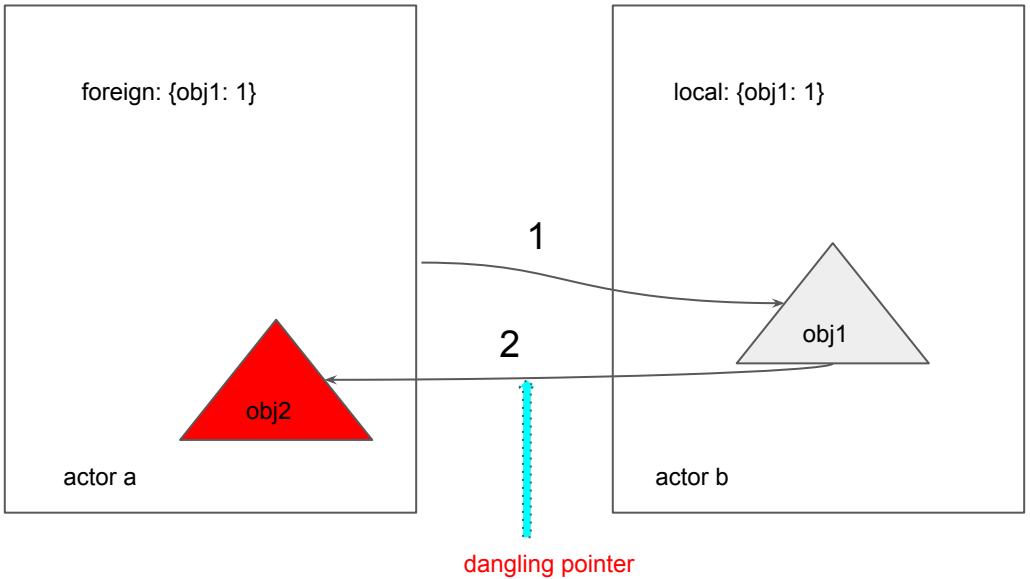
Example of extending objects after receiving (2)




```
class A {
  def method(o : Obj) {
    o.setNext(new Obj)
    o
  }
}

class B {
  a = new A
  obj = new Obj
  obj = get a.method(obj)
  obj.getNext()
}
```

actor **a** extents **obj1** with **obj2** after receiving

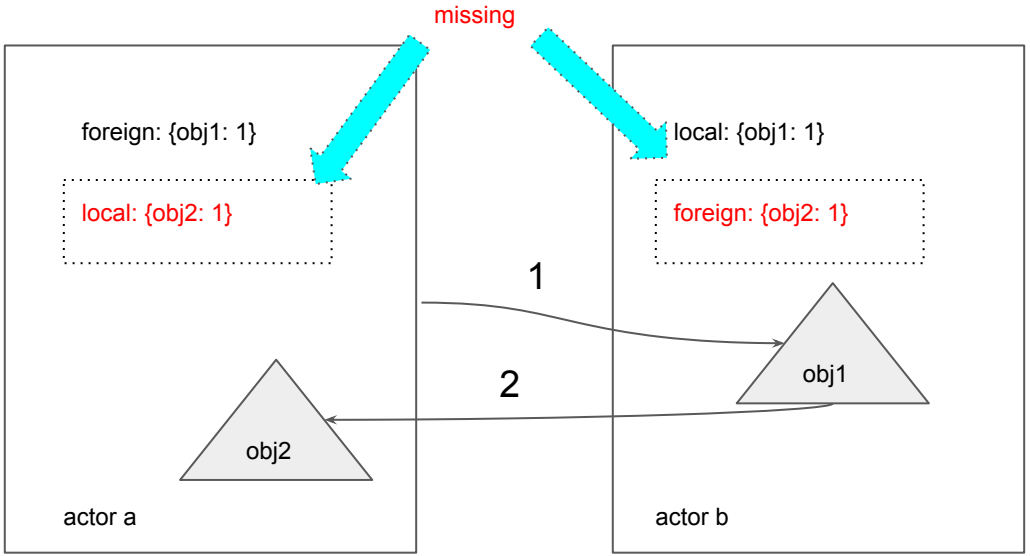
Example of extending objects after receiving (3)




```
class A {
  def method(o : Obj) {
    o.setNext(new Obj)
    o
  }
}

class B {
  a = new A
  obj = new Obj
  obj = get a.method(obj)
  obj.getNext()
}
```

GC in actor **a** would collect **obj2**, because it's not captured in the **local** set

Example of extending objects after receiving (4)



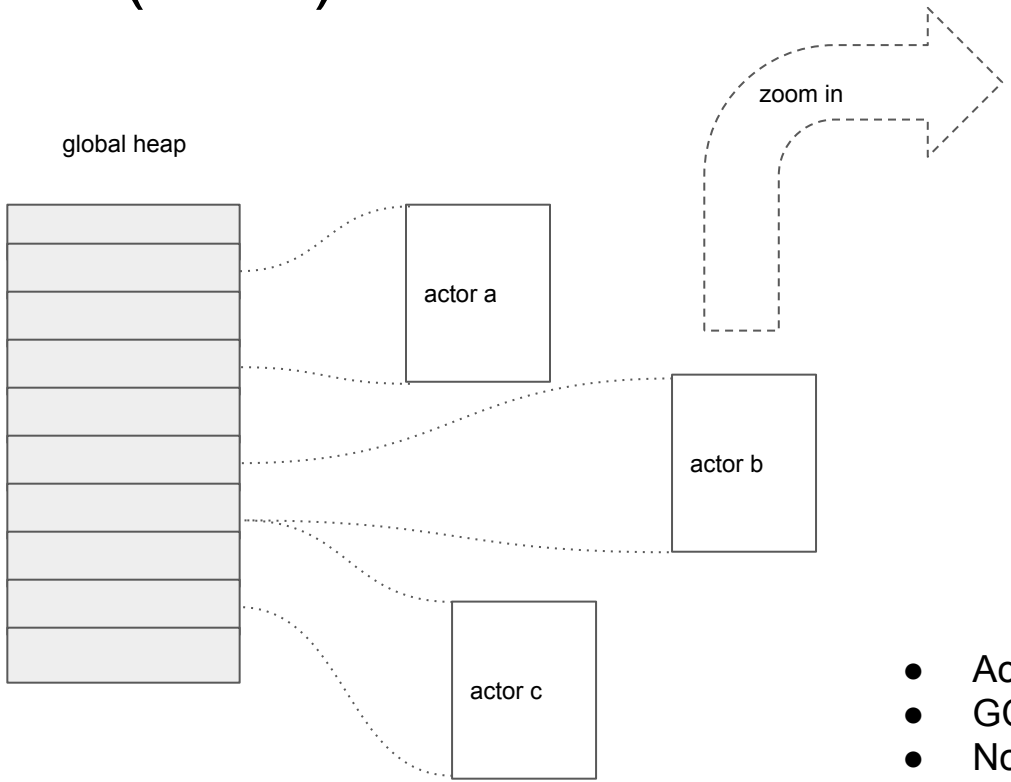

```
class A {
  def method(o : Obj) {
    o.setNext(new Obj)
    o
  }
}

class B {
  a = new A
  obj = new Obj
  obj = get a.method(obj)
  obj.getNext()
}
```

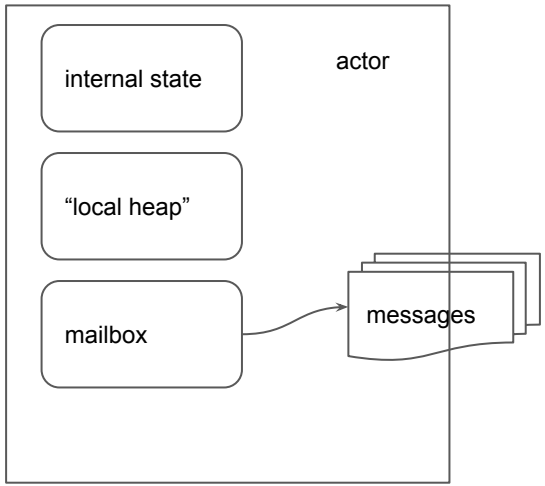
The correct state to keep **`obj2`** from being collected

How Object GC in other actor-based systems work

- Scala (Akka)
- Erlang

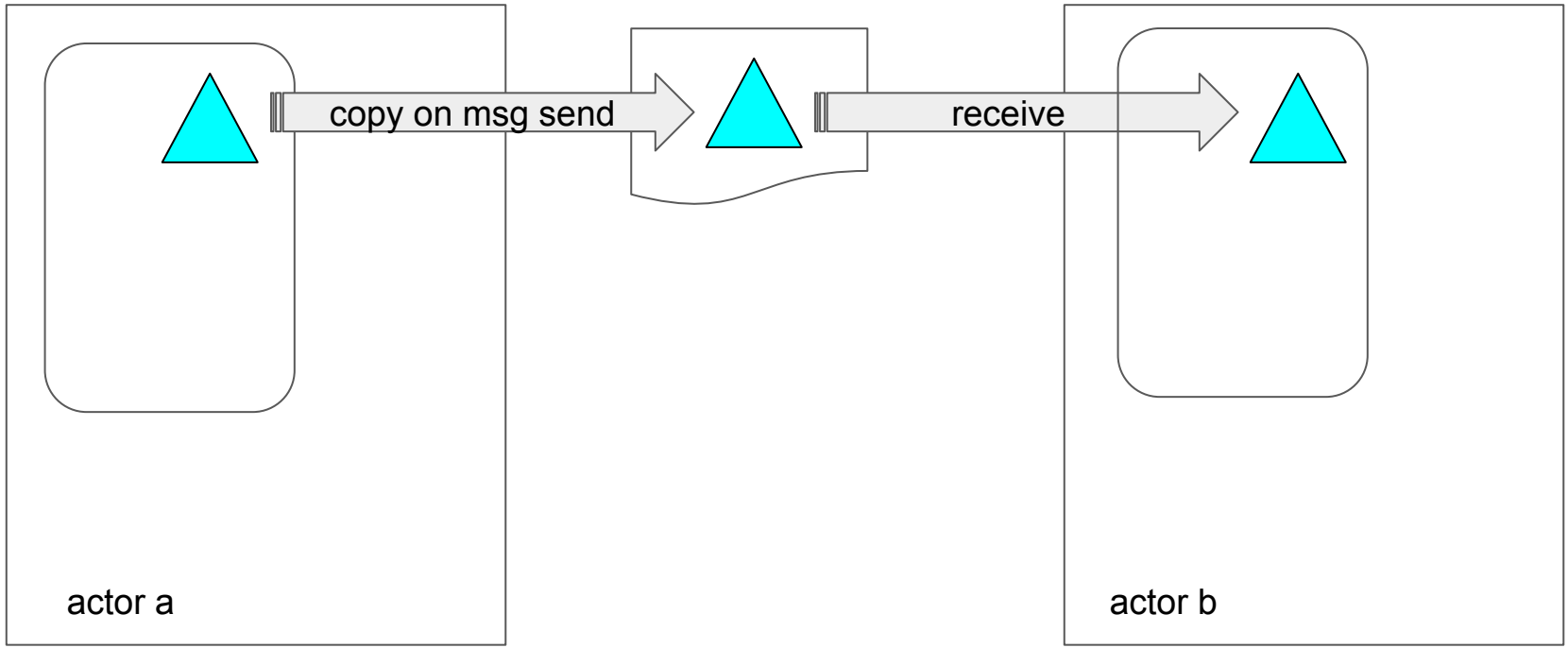
Scala (Akka)



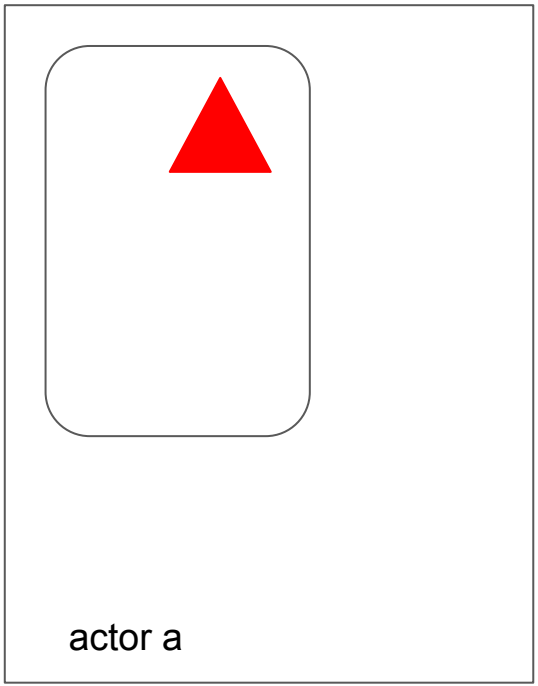


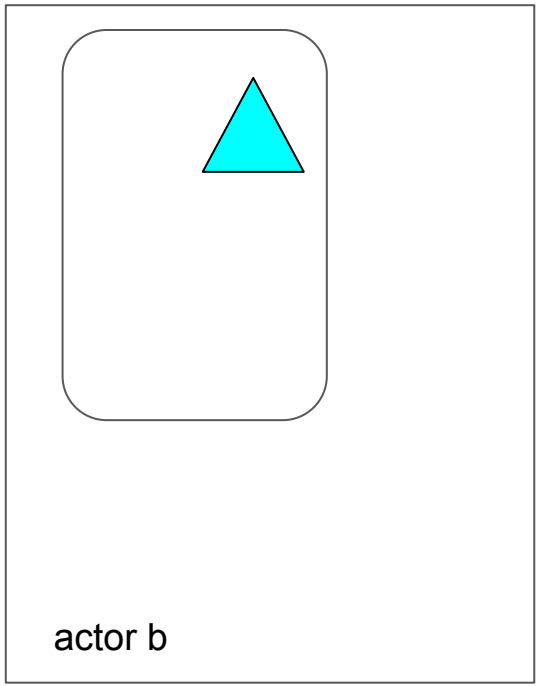
- Actors are mapped into global heap
- GC is performed on global heap
- No parallel GC on per-actor level

Erlang (1)



Erlang (2)





- Two independent copies
- Parallel GC without interference

Comparison on Object GC

- Scala (Akka)
  - global GC can’t take advantage of the isolation of actors
- Erlang
  - local GC, but sharing large objects is expensive
- Encore
  - local GC, sharing is cheap, but has GC issues in concurrent access to shared passive objects...

Proposed Solution: Shared Object

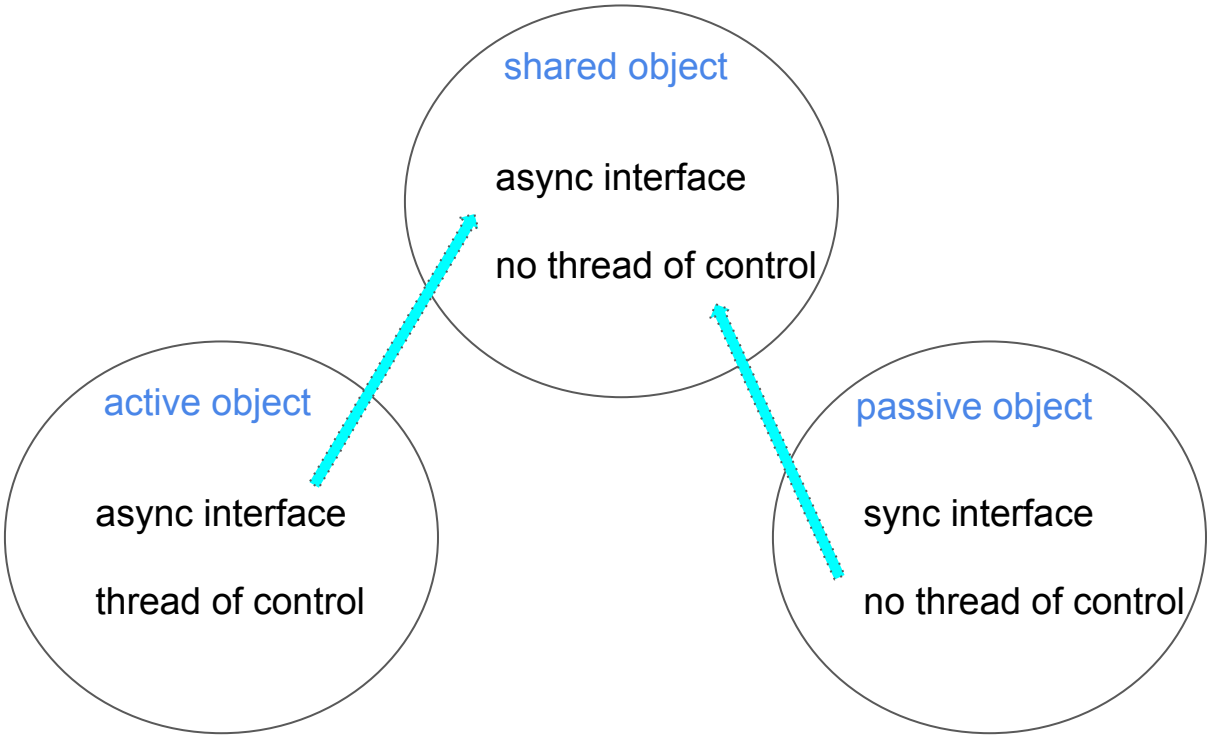
Isolated container, supporting a limited form of sharing and contention

- Support concurrent access to **its** encapsulated passive objects
- Support for parallel method execution

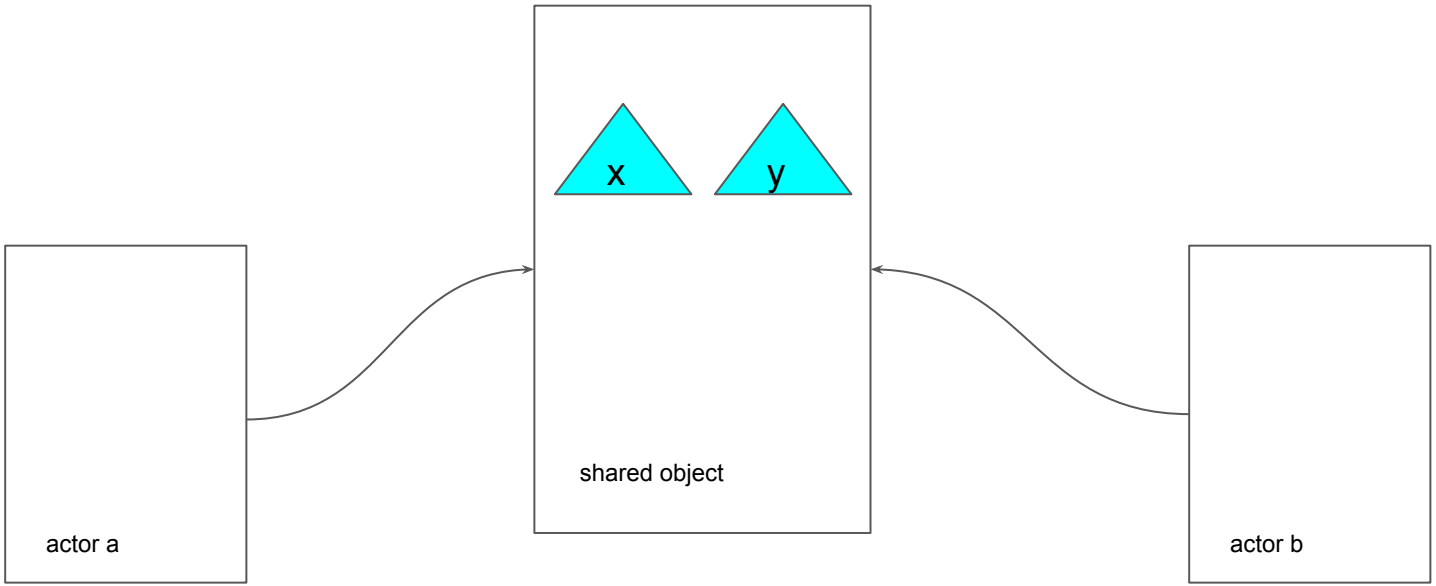
Important GC points

- No global sharing of mutable state, i.e., still parallel GC
- Only place where GC is contended is inside the heaps of shared objects

Active -- Shared -- Passive

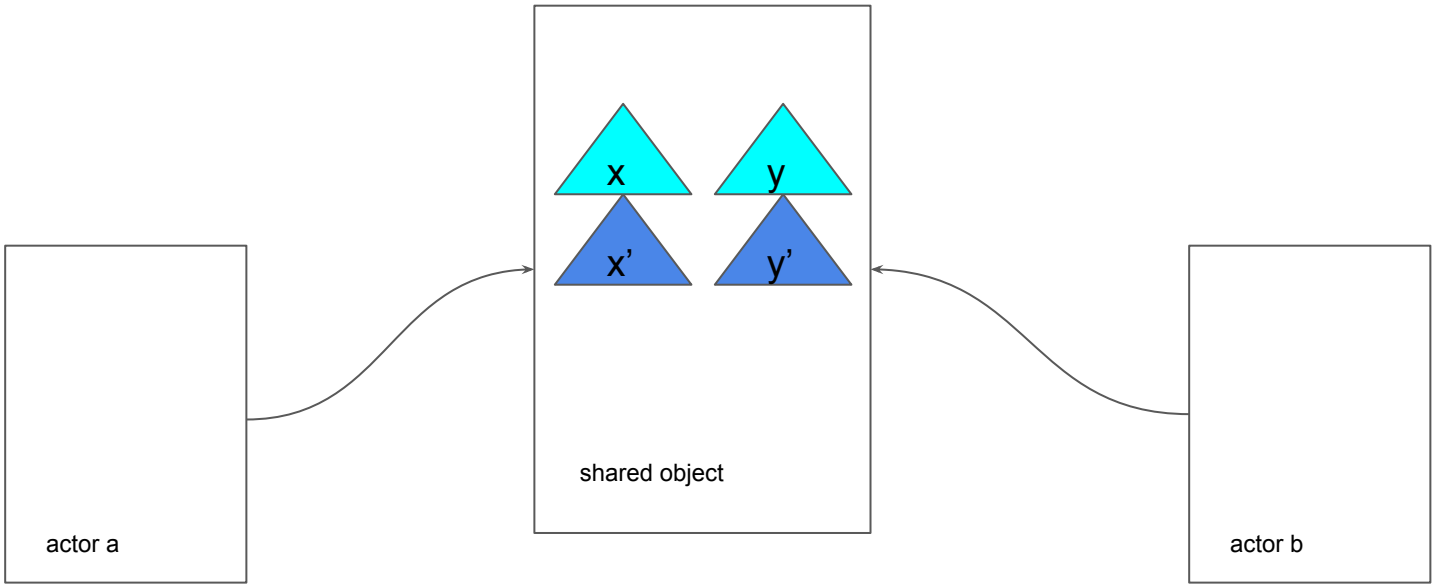


Example of Shared Object (1)



actor **a** and **b** have access to a shared object, that has two disjoint fields, **x** and **y**

Example of Shared Object (2)



two actors could extend mutable state on shared object's heap, synchronously and concurrently

Implementation Space

- Actor
  - Shared Object has the same interface as actor
  - high latency due to async operations
- Semi-actor
  - the caller actor acquires the Shared Object if it's not acquired already
  - the caller actor do operations synchronously
  - the caller actor insert the operations to Shared Object's mailbox it's already acquired
  - (Similar to Queue Delegation locking.)
- STM
  - Wrap each operation inside a transaction
  - get parallelism when operations don’t overlap

...

Summary

- GC trade-offs: local vs global heap, copy vs pass-by-ref
- Sharing mutable passive objects could cause GC issues in local + pass-by-ref
- Shared Objects (SO) enables shared states in its heap
- Actors could collaborate on data structures in parallel via SO
- Large implementation space: actor, QD locking, STM…

Questions, Comments

&

Suggestions