The Actor Model

Radu Nicolescu
Department of Computer Science
University of Auckland

20 March 2019//26 March 2019

Actor model (Wiki)

- The Actor model is a model of message-based concurrent computation which treats "actors" (aka "agents") as the universal primitives
- In response to a message that it receives, an actor can
  - make local decisions
  - create more actors
  - send more messages
  - (change state) determine how to respond to the next message received
- There is no assumed sequence to the above actions
- Theoretically, it could take an unbounded time for a message sent to be received – but this will eventuate (guaranteed)!

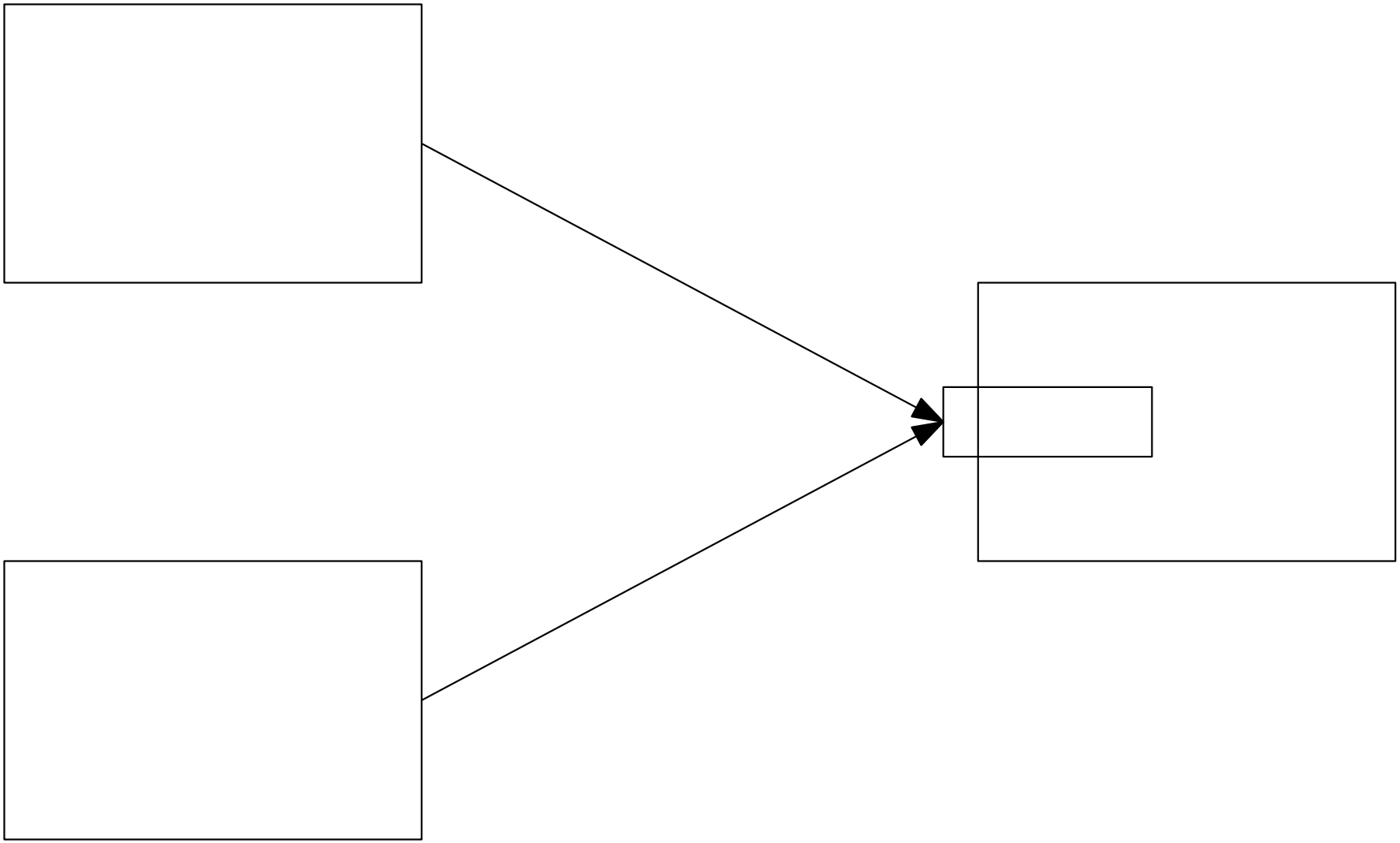
Actor model and theory

- Actor model was originally inspired by the laws of physics
- Major early contributions: Carl Hewitt, Will Clinger, Gul Agha, Henry Baker, ...
- Actor model on Wikipedia: `http://en.wikipedia.org/wiki/Actor_model`
- The Actor model raises interesting theoretical questions
- Fairness, unbounded non-determinism
- Controversy: can an an unboundedly nondeterministic program solve problems not solvable in the Turing framework?
- Unbounded nondeterminism on Wikipedia: `http://en.wikipedia.org/wiki/Unbounded_nondeterminism`

Actor frameworks

- Erlang (Ericsson, telecommunications)
- Java: Scala, Akka
- F#: MailboxProcessor
- F# & C#: Akka.NET, Reliable Actors, Orleans (+ Orleankka), Unbounded channels

Actors – bird's eye view



MailboxProcessor – bird's eye view

- Actor = an async queue of typed messages + an associated...
- "behaviour" function which asynchronously processes posted messages – single-threaded but not necessarily same thread
- `MailboxProcessor<'T>` : type of actor and its queue
- `MailboxProcessor<'T> -> Async<unit>` : associated function
- The function returns `Async<unit>`, so MailboxProcessor actors have NO final value
- All values, including any intended final "result" (if any), are exchanged by asynchronous messaging
- Termination: how to terminate an actor system or detect its termination?

MailboxProcessor (Actor/Agent)

- The actor encapsulates an "inbox" message queue that supports multiple-writers and a single reader (the actor itself)
- Writers can send one-way messages to the actor by using the Post method and its variations
- Writers can send two-way messages (i.e. messages which request replies) to the actor by using variations of the PostAndReply method
- Actors can receive messages using the Receive method and its variations (with optional timeouts)
- Actors can also scan through all their available messages using the Scan method and its variations

MailboxProcessor – messages

- The message types (normally use discriminated unions)

```
1  type Msg1 = ...    // one-way to-actor messages
2
3  type Reply = ...   // one-way from-actor messages
4
5  type Msg2 = ... AsyncReplyChannel<Reply> ...
6                     // two-ways send/reply messages
7
8  type Msg = | Msg1 | Msg2        // all to-actor messages
```

MailboxProcessor in a nutshell

- Ideally, all actor's interactions with the external world, including its "result", are performed by send/reply messages (unless the actor has side-effects)
- To asynchronoulsy post a one-way message, (msg1:Msg1), to actor a:

```
1 a.Post msg1
```

- To receive a message (msg:Msg), via an asynchronous awaited call (with optional timeout):

```
1 let! msg = inbox.Receive ()   // Async<Msg>
```

MailboxProcessor in a nutshell

- To post a two-ways message, (msg2:Msg2), to actor a, and wait for the reply (rep:Reply)
- Blocking call (with optional timeout)

```
1 let rep= a.PostAndReply(fun repchan->msg2)
2           // Reply
```

- Asynchronous awaited call (with optional timeout)

```
1 let! rep= a.PostAndAsyncReply(fun repchan->msg2)
2           // Async<Reply>
```

- You have to construct msg2, by including the reply channel repchan: AsyncReplyChannel<Reply>, which the runtime system gives as argument to your function

MailboxProcessor in a nutshell

- The static factory function Start is a typical way to create and start an actor:

```
1 let a = MailboxProcessor.Start
2                 (fun inbox -> async { ... })
```

- Your task is to define Start's parameter: a "constructor" function `(MailboxProcessor<Msg> -> Async<unit>)`
- The "inbox" parameter represents the input message queue
- You do NOT create the "inbox" argument: the runtime invokes your "constructor" function with the proper argument
- The created actor has the same type as its "inbox", and its actual body has type `Async<unit>`

MailboxProcessor construction details

1. Factory static method Start(**fun** ...) (create & start)

```
1  let actor = MailboxProcessor.Start(fun inbox ->
2                  async {
3                      ...
4                      let! msg = inbox.Receive()
5                      ...
6                  } )
```

2. Constructor MailboxProcessor(**fun** ...) ,
   then instance method Start()

```
1  let actor = new MailboxProcessor(fun inbox ->
2                  // as above
3                  )
4  ... //  more setup...
5  actor.Start()
```

MailboxProcessor construction

- This is an interesting case of IOC (Inversion of Control)
- Aka the Hollywood principle: "Don't call us, we'll call you"
- You never call the function given as parameter to the MailboxProcessor factory method Start: the runtime system calls it
- Your "constructor" function automatically receives an inbox argument, pointing
- The same principle applies to the function parameter of PostAndReply...

Sample Actor: mutable counter

```
1  let agent1 = MaiboxProcessor.Start(fun inbox ->
2    let count = ref 0                          // notice the ref
3    async {
4      while true do
5        let! msg = inbox.Receive()
6        incr count
7        printfn "agent1 total=%d messages" !count
8    } )
9
10 ["the"; "quick"; "brown"; "fox"]
11 |> List.map agent1.Post
```

- Quiz: what is agent1's type, i.e. what is the inferred 'T?

Sample Actor: mutable counter

- Here, a mutable counter is implemented via a ref, not mutable, construct
- In this async monad, this mutable counter is part of a closure object
- Ref mutable variables are implemented on the heap
- Recall the old famous funarg problem: essentially, how to implement closures
- And the difficulties that some languages still have to promote stack variables to closures
- For example, Java's requirement that local free variables used by local functions must be final

Sample Actor: recursive counter

```
1  let agent2 = Agent.Start(fun inbox ->
2    let rec loop count =                    // loop is recursive
3      async {
4        let! msg = inbox.Receive()
5        printfn "agent2 total=%d messages" (count+1)
6        return! loop (count+1)    // ≡ tail recursion
7      }
8    loop 0
9    )
10
11 ["jumps"; "over"; "the"; "lazy"; "dog"]
12 |> List.map agent2.Post
```

- "Recursive tail calls": **return**! optimised by trampolines!
- The stack frame for n is replaced by the stack frame for n+1

Sample state-machine actor

- The body of your constructor function must return an async
- That async can: (1) *terminate*, (2) *loop* (potentially infinite) or (3) return another async (forming a chain of asyncs)
- Such a chain of asyncs can represent a chain of state changes!
- Our sample actor toggles between two states, active and inactive, defined by a pair of inner mutually recursive functions.
- In active state, it adds received numbers towards a grand total; in inactive state, it ignores the add messages.
- In both states, if requested, sends back the current total, via the automatically created response channel coming from its caller

Sample state-machine actor

```
1  agent.Post (Add 10)                    // → active 10
2
3  agent.Post Toggle                      // → inactive 10
4  agent.Post (Add 20)                    // → inactive 10
5
6  agent.Post Toggle                      // → active 10
7  agent.Post (Add 30)                    // → active 40
8
9  let n = agent.PostAndReply (           // calls & waits
10     fun ch -> (Get ch))                // ch is the automatically created
11                                        // response channel
12
13 printfn "got %d" n                     // → got 40
```

- Magical IOC: you do not create the reply channel, the system will create it – just provide its place as a function parameter

Sample state-machine actor

```
1  type Message =
2       | Toggle
3       | Add of int
4       | Get of AsyncReplyChannel<int>
5
6  let agent = MailboxProcessor<Message>.Start (
7      fun inbox ->
8        let rec active n = async { ... }
9        and inactive n = async { ... }
10       active 0 )                    // initially active with count 0
```

- Two mutually recursive functions, active and inactive, record the state, without changing any variable...
- "Tail calls": **return**! optimised by trampolines!

Sample state-machine actor

```
1   let rec active n = async {
2     printfn "active %d" n
3     let! msg = inbox.Receive()
4     match msg with
5       | Toggle -> return! inactive n
6       | Add m -> return! active (n + m)
7       | Get ch -> ch.Reply n; return! active n
8     }
9
10  and inactive n = async { ... }
```

- Note: **return**! "tail calls"

Sample state-machine actor

```
1   let rec active n = async { ... }
2
3   and inactive n = async {
4     printfn "inactive %d" n
5     let! msg = inbox.Receive()
6     match msg with
7       | Toggle -> return! active n
8       | Add _ -> return! inactive n
9       | Get ch -> ch.Reply n; return! inactive n
10    }
```

- Note: **return**! "tail calls"

Sample for async scalability

- Massive test: 100,000 asyncs

```
1  let asyncs =
2    [ for i in 0 .. 100000 ->
3        async {
4            Async.Sleep(1000)
5            if i % 10000 = 0 then
6                printfn "async %d" i
7    } ]
8
9  Async.Parallel asyncs |> Async.RunSynchronously
```

- A couple of secs

Sample for actor scalability

- Massive test: 100,000 pinged actors

```
let agents =
  [ for i in 0 .. 100000 ->
    MailboxProcessor.Start(fun inbox ->
      async {
        while true do
          let! msg = inbox.Receive()
          if i % 10000 = 0 then
            printfn "agent %d got message '%s'"
                i msg
  } ) ]

for agent in agents do
  agent.Post "ping!"
```

- A couple of secs

More about F# actors

- Please *actively* study all the given samples...
- ... including the stress tests (not covered in this handout), e.g.
  - a single actor that can process about 500,000–1,000,000 simple messages per second
  - spawning and killing 1,000,000 actors at the rate of one each 10–12 microseconds
- ... refer as required to the MSDN pages

Unbounded non-determinism

- An actor which, theoretically, can return/print any natural number and is also guaranteed to terminate (thus, no non-ending loop possible)
- Debate: can this have a practical implementation, which will do the same, barring accidents?
- Note: it is easy to define a probabilistic system, by repeated coin flipping, which can return any natural number - but this system may never terminate
  - an event with probability 0.0 can still happen in an infinite event space

Unbounded non-determinism – F#

```
1  let rec und = MailboxProcessor.Start (fun inbox ->
2      let rec loop count =
3          async {
4              und.Post 1
5              let! msg = inbox.Receive ()
6              match msg with
7              | -1 -> printfn "%d" count
8              | _ -> return! loop (count+1)
9          }
10     loop 0)
```

- Send it message -1 – to print the current counter and exit
- Self messages 1 increment the internal counter and loop
- -1 is guaranteed to arrive, but no bound on its delivery
- Theoretically, any number could be printed!

How to detect termination

- Detecting the termination of a distributed systems is a non-trivial problem
- We suggest three ways - but all require some cooperation from the concerned actors
  - Using low level system events
  - Using Task sources (Task unit!)
  - Using a dedicated termination actor
- Interlocked "magic": fastest thread-safety for a few arithmetic operations
  - Hardware wait-free, e.g. using the Intel LOCK prefix

Termination with system events

```
1  let acount = ref 0   // total count of actors
2  let adone = new AutoResetEvent (false)   // main waiting
3
4  let act i = MailboxProcessor<int>.Start(fun inbox ->
5    async {
6      let! m = inbox.Receive ()
7      ...
8      if Interlocked.Decrement acount = 0 then
9        adone.Set () |> ignore   // release main
10   })
```

```
1  // main waiting
2  adone.WaitOne () |> ignore
```

Termination with Task sources

```
1  let acount = ref 0  // total count of actors
2  let adone = TaskCompletionSource<bool> ()
3
4  let act i = MailboxProcessor<int>.Start(fun inbox ->
5    async {
6      let! m = inbox.Receive ()
7      ...
8      if Interlocked.Decrement acount = 0 then
9        adone.SetResult true
10   })
```

```
1  // main waiting
2  adone.Task.Wait ()
```

Termination with dedicated termination actor

- Messages of the termination actor

```
1 type AdoneMSg =
2       | Done of int
3       | Subscribe of AsyncReplyChannel<bool>
4
5 let acount = ...   //   exact expected number of actors
```

Sample termination actor – accepts multiple subscribers

```
1  let adone =
2    MailboxProcessor<AdoneMSg>.Start(fun inbox ->
3    let rec loop count
4          (chs: list<AsyncReplyChannel<bool>>) =
5      async {
6        let! m = inbox.Receive ()
7        match m with
8        | Done i ->
9          if count > 1 then return! loop (count-1) chs
10         else chs |> List.iter (fun ch -> ch.Reply true)
11              return! loop 0 []
12       | Subscribe ch ->
13         if count > 0 then return! loop count (ch::chs)
14         else ch.Reply true
15              return! loop 0 []
16     }
17   loop acount [])
```

Sample actors and main

```
let act i = MailboxProcessor<int>.Start(fun inbox ->
  async {
    let! m = inbox.Receive ()
    ...
    adone.Post (Done i)
  })
```

```
let isdone = adone.PostAndAsyncReply Subscribe
...
Async.RunSynchronously isdone |> ignore
```