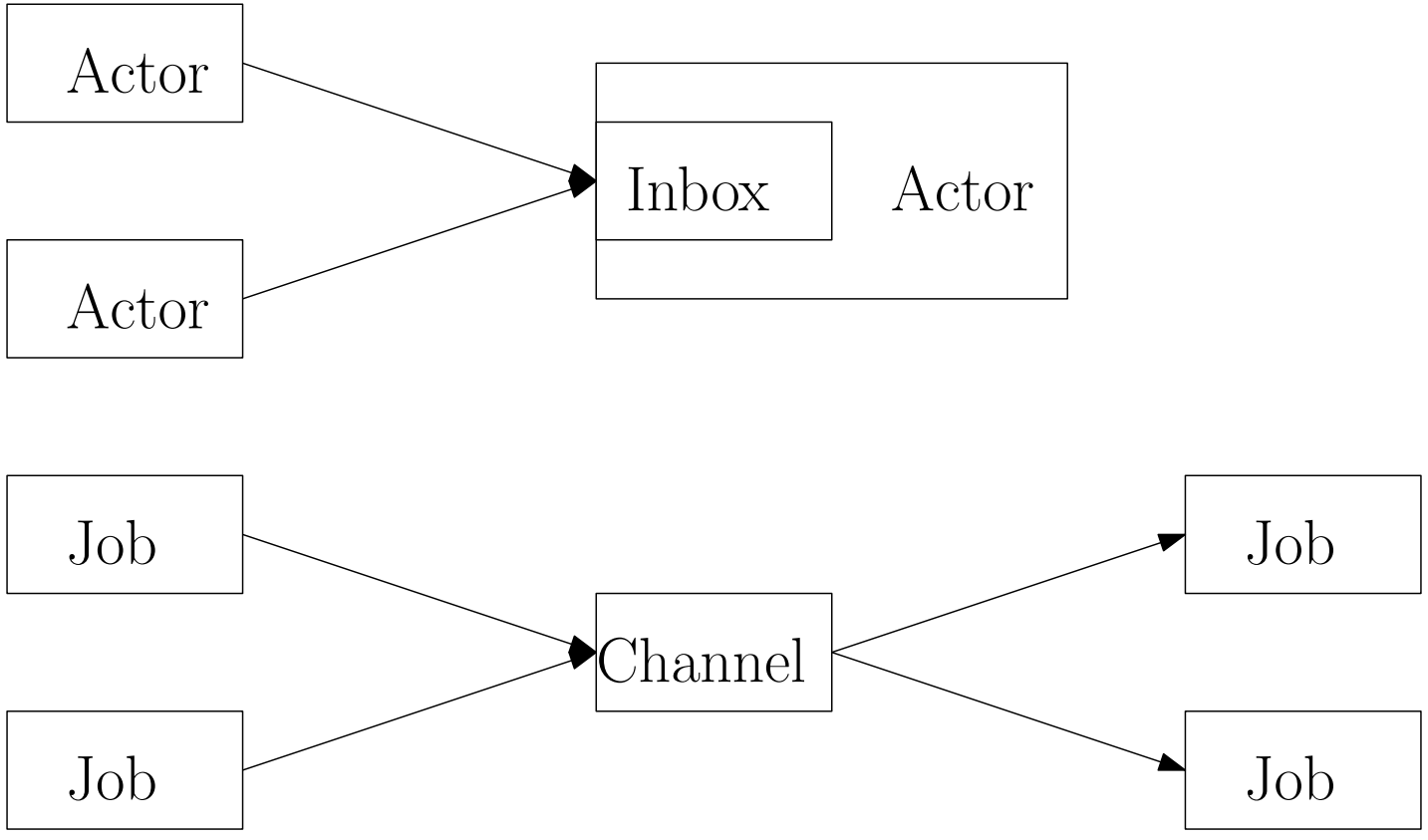
# Actors and Channels – Part II

Radu Nicolescu
Department of Computer Science
University of Auckland

2 April 2019

1. Actors vs Channels
2. Multiple mailboxes
3. Selection
4. Backpressure
5. Supervision and monitoring
6. Misc

# Actors vs Channels – in a nutshell



# Actors vs Channels – heavy- vs light-weight threads

Both Actors/Mailbox and CML/Hopac are models of message passing concurrency, between asynchronous tasks

Major differences:

- Different task weights:
  - Actors use async tasks, directly mapped to heavyweight threads from the common thread pool, which are costly
  - Hopac uses a huge numbers of very lightweight job tasks, served by their own thread pool
  - Unlike usual tasks or threads, many lightweight jobs can be logically blocked without affecting the system
- Different basic message passing primitives: *next slide*

# Actors vs Channels – async vs. sync message passing

- Different task weights: *prev slide*
- Different basic message passing primitives – async vs. sync:
  - Actors are based on asynchronous post/receive primitives for actors, supported by mailbox-type queues
  - Basic Hopac is based on synchronous channels, essentially rendezvous primitives which do NOT need queues
  - However, one can build sync communications on top of mailboxes and async communications on top of channels ...

    `https://github.com/Hopac/Hopac/blob/master/Docs/Actors.md`

# Actors vs Channels – performance

- Where applicable (NOT everywhere), queue-less rendezvous operations are faster and less demanding on system resources (no disposables, no heap allocation, ...)
- Otherwise, for apps that genuinely require queues, queue-less rendezvous operations may show less performance
- Hopac jobs are designed and optimized to scale as the number of such relatively independent lightweight elements is increased, e.g. on parallel systems (for apps that do NOT require queues)
- However, there is NO direct support for distributed computing, e.g. clusters or clouds, where Actors shine – are the "gold standard"

# Actor with multiple mailboxes?

- Not part of the theoretical model, but...
- Many practical implementations allow this “abuse”...
- Or don’t disallow it
- Mailbox
  - Is safe for many writers (senders)
  - But requires one single reader (receiver)
- How to solve it?

# Actor receiving from multiple mailboxes!

- Briefly: do NOT start helper actors, just use their mailboxes

```
1  let a = new MailboxProcessor<int> (fun inbox ->
2          async { () })
3  let b = new MailboxProcessor<int> (fun inbox ->
4          async { () })
5
6  let c = new MailboxProcessor<int> (fun inbox ->
7      let rec loop () = async {
8          let! x = a.Receive ()   // first, await a
9          let! y = b.Receive ()   // then, await b
10         ...   // zips a and b
11         return! loop ()
12         }
13     loop ())
```

- Full sample: F#-Mailbox-MultiZip.linq

# Selection in Hopac

- Selective message passing means that a form of choice or disjunction between alternatives is supported
- Choosing between incoming channels – i.e. receiving messages

```
1  let! m =
2      Alt.choose [
3          ch1;
4          ch2;
5          ...
6      ]
```

- Full sample: F#-Hopac-MultiSelect.linq

# Selection in Hopac

- More generally, choosing between a list of arbitrary events – receiving messages, sending messages, timeouts, ...

```
1  do! Alt.choose [
2          event1 ^=> fun () -> ...
3          event2 ^=> fun () -> ...
4          ...
5      ]
```

- Useful guarantee (in Hopac, not in all channel frameworks): the lexically first succeeding event is always chosen
- E.g. useful for prioritising channels or other events ...

# Selection in Actors?

- One inbox solutions
  - Mailbox: scanning (filtering) for the first message with a given format or property – w/ timeout option

```
1 let t = inbox.Scan (fun msg -> ...)
```

  - Akka: stashing / unstashing less priority messages for later processing
- Several inboxes (see prev. slides) – more complex solutions
  - Full sample (one version): F#-Mailbox-MultiSelect.linq

# Backpressure aka Throttling

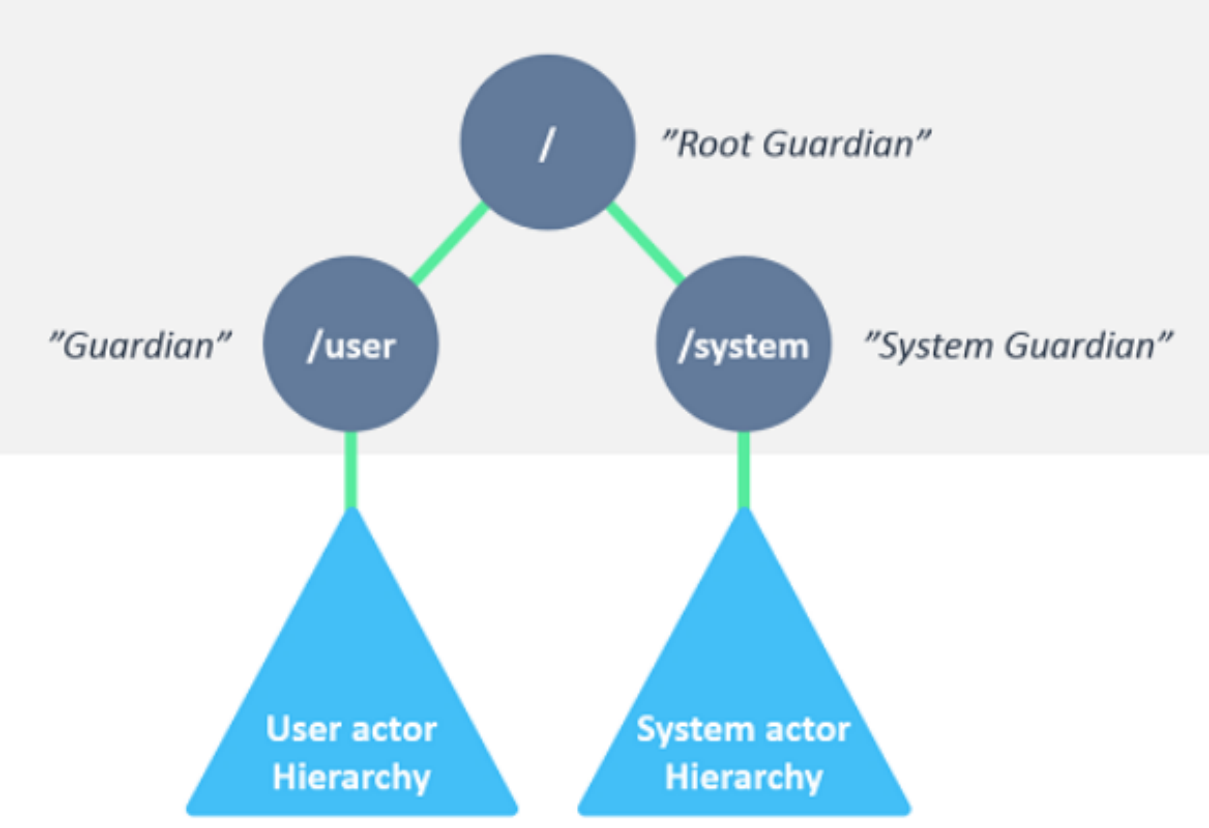
- Problem: Fast Producer (sender, source, publisher), vs. Slow Consumer (receiver, sink, subscriber)
- Free backpressure for 0-size buffers channels
  - Writer awaits for space
  - Full sample: F#-Hopac-Backpressure.linq

# Backpressure aka Throttling

- More solutions for C# bounded buffers channels
  - FullMode = DropNewest : deletes the newest item
  - FullMode = DropOldest : deletes the oldest item
  - FullMode = DropWrite : drops the item being written
  - FullMode = Wait : await for space
- Actors Akka: similar solutions, but also even more...
- Actors F# Mailbox: does it offer a basic throttle?
  - Full sample: F#-Mailbox-Backpressure.linq

# Supervision hierarchy (Akka.NET doc)

- To read: https://getakka.net/articles/concepts/supervision.html
- Supervisors delegate tasks to subordinates (children)



# Supervisor directives (choices)

- Subordinates may fail, i.e. throw unhandled exceptions, but this does NOT automatically terminate them!
- Instead, failure messages are sent to supervisors – into dedicated mailboxes!
- Supervisor directives decide responses:
  - Resume the subordinate(s), keeping internal state(s)
  - Restart the subordinate(s), clearing internal state(s)
  - Stop the subordinate(s) permanently
  - Escalate the failure to the next parent in the hierarchy
  - these are affected by the supervision strategy (next slide)

# Supervision strategies

- Supervision strategies:
  - define mappings from exception types to supervision directives i.e. to resume, restart, stop, escalate
  - limit how often children are allowed to fail before terminating them
- Two classes
  - OneForOneStrategy stategy: applies only to the failed child
  - AllForOneStrategy stategy: applies to the failed child and all its siblings
- BackoffSupervisor built-in pattern
  - implements the exponential backoff supervision strategy
  - starts a child actor again when it fails
  - each time with a growing time delay between restarts

# Actor Termination

- Normal termination
- Supervisor decided termination (stop)
- Stop () method
- PoisonPill message (sent to the mailbox)
- Kill () method
  - triggers an ActorKilledException
  - which is handled by the supervisor
  - and usually ends in termination

# Monitoring

- Each actor may monitor any other actor for termination
  - Watch (...), Unwatch (...) methods
- Termination of the watched actor is implemented by
  - Terminated message sent to watcher's mailbox
  - which triggers a DeathPactException
  - and is then handled by the watcher's supervisor

# Actor distribution – decentralized P2P networks

- Remote deployment: usually on fixed network
- Cluster deployment: fault-tolerant and elastic network
- Delivery: at-most-once, at-least-once, exactly-once
  `https://www.infoq.com/articles/no-reliable-messaging`
- Actor code migration?
  - C# Compiled actor code must already be present on the remote sites
  - F# Code can be transported as data via code quotations – generalisation of .NET ASTs (abstract syntax trees)
    `<@ ... @>`
    - Quotation sample: Quotations F#-quotations.linq