# Actors in the Small

Making Actors more Useful

Bill La Forge
laforge49@gmail.com
@laforge49

# Actors in the Small

I. Introduction
II. Making Actors Fast
III.Making Actors Easier to Program
IV. Tutorial

# I. Introduction

- What is an Actor?
- Actors are SLOW
- Why are Actors Slow?
- Actors can be FAST

# What is an Actor?

- Actors are very light-weight threads, lacking even their own stack
- Actors process events they receive them and send events to other actors
- Actors process events one at a time, so processing is both thread-safe and atomic
- Actors are a good alternative to threads and locks, easier to scale vertically and easier to debug.

# Actors are SLOW

- Akka, one of the faster actor frameworks, has a throughput of 20 million events per second when run on a box with 45 CPU's
- That is fast enough, so long as events are not low level
- Parsing is not a good use of actors, not within a web server anyway

# Why Actors are Slow

- Message passing is always between threads, which is slow, especially when the destination actor is idle
- Using events under load often raises an out of memory exception unless flow control is implemented by the application

# Actors can be FAST

- JActor has a throughput of between 75 million and 1 billion messages per second when run on an i5 with 4 hardware threads
- The scope of a technique (actor programming) is often limited by its speed
- JActor is fast enough to use for things like parsing and CometD
- This is Actors in the Small

# II. Making Actors Fast

- Message buffering
- Single threaded
- Two-way messages
- Decoupling actors and mailboxes

# Message Buffering

- Messages to be passed to an actor are buffered and passed as a group when the actor becomes idle (Message buffering is a common technique in flow-based programming)

# Single Threaded

- Frameworks like node.js and Twisted achieve high throughput by performing all event processing on a single thread
- Except as required by the application, JActor never sends a message to an idle actor, rather the source actor commandeers the destination actor and processes the message itself, synchronously

# Two-Way Messages

- Flow control is implicit with 2-way (request / response) messaging, and applications behave more reasonably under load
- Synchronous processing (call / return) more easily maps onto request / response messages than onto events
- Unlike events, request processing is not atomic if that processing sends any requests

# Decoupling Actors and Mailboxes

- In JActor, mailboxes are light-weight threads, actors are not
- Every actor needs a mailbox; a mailbox can be used by many actors
- Request / response messages passed between actors with the same mailbox are processed synchronously, e.g. as a method call / return.

# III. Making Actors Easier to Program

- Asynchronous message passing (sending messages) requires callbacks
- Synchronous message passing (method call / return) does not

# Sending a Request Message

- When sending a request to another actor, a callback is used to receive the response
- For asynchronous responses, the callback is invoked AFTER the sending method returns
- For synchronous responses (when only a single thread is involved), the callback is invoked immediately.
- You can not always predict which responses will be synchronous or asynchronous, e.g. when accessing a cache and an item is not present

# Sending a Response Message

- When a request is received by an actor, a callback is also passed
- Responses are returned by invoking the callback
- Exactly one response must be returned for each received request
- The callback sent by the source actor with a request is often not the same as the callback received by the destination actor due to intermediation by JActor internals

# Special Request Message Types

- Three common types of special requests are
  (1) synchronous,
  (2) initialization and
  (3) concurrent
- These requests can be sent without a callback via a method call—they are always processed synchronously and the response is the return value
- An actor which receives these requests is not passed a callback and can not send requests asynchronously to other actors

# (1) Synchronous Requests

- An actor may call another actor with a synchronous request only if both actors share the same mailbox; otherwise the request must be sent (with a callback)
- Synchronous requests, like regular (asynchronous) requests, are processd by the target actor with full thread safety.
- Unlike regular requests, synchronous request processing is atomic.

# (2) Initialization Requests

- An actor will not process an initialization request after processing any other kind of request
- Thread safety is entirely the responsibility of the actor that sends the initialization requests—it must ensure that such requests are sent one at a time

# (3) Concurrent Requests

- Concerency requests can and will be processed at the same time that the receiving actor is processing other requests—there is no inherent thread safety
- Concurrent request processing must only access immutable and concurrent data structures
- Thread safety is entirely the responsibility of the applicatin logic used to process the request

# IV. Tutorial

JActor by Example

https://github.com/laforge49/JActor/wiki/Examples