Main Summary:

- *Java is a OO programming language with built in support for concurrent programming by means of threads but there exists high risk in using threads as can be seen by stepping through my initial part of presentation.*
- *On the other hand Scala provide a safer programming option for concurrency which boosts the performance of the program using Actors model.*
- *Scala is compatible with Java that means that all the existing Java library can be used by Scala.*
- *Years of research behind the implementation of the JIT makes Java performs quite similar to C/C++ and the programming model adopted in Scala is much closer to the traditional object oriented paradigm used by most of the software developers.*
  - *The actor model allows to write complex parallel and distributed programs in a very simple way.*
  - *The overhead introduced by the environment (due to the management of the mailboxes) can deeply impact the performances in several kind of application contexts.*
  - *It is widely used in multi-agent systems (MAS) systems because, mailboxes provides a simple way to build and coordinate distributed components.*
- *If you are looking for performances, SCALA gives you several communication primitives (synchronous, asynchronous communication and collective operations) and remains the best choice for computational intense tasks.*
- *Actors are used for fast prototyping in environments (like Web Services) I/O intensive applications.*

Concurrency In Java And Actor-Based Concurrency Using SCALA.

by T. Sravan Kumar.

CSCI-5448 Spring 2011.

Prof. Kenneth Anderson.
The University Of Colorado, Boulder.

Contents

- Introduction to Concurrency.
- Brief History of Scala.
- Actor Based Concurrency Model.
- Main Points of the Actor Model.
- Scala Actor Properties.
- Programming Abstractions.
- Concurrent Programming in Scala.
- Performance Evaluation.
- Conclusion.
- Q and A.

Introduction:
Brief History of Concurrency

- Sequential to Concurrent: You can't run a single-threaded application on a multi-core processor and expect better results.
  - Java Language supports threads which leads to concurrency and Parallelism.
- Multiple contexts/program counters running within the same memory space.
- All objects can be shared among threads in Java.
- Fundamentally nondeterministic in nature.
  - Not a Hard Real Time system but a Best effort system.

Single Processor Vs MultiCore CPU Concurrency

- Multitasking on single processor – Old Method.
  - I/O wait vs. CPU
  - Background processing in UI
- Multicore CPUs (true parallelism)
  - Large units of standalone work
    - Processing client requests on server
    - data processing (divide into smaller independent units)
  - I/O wait
- "Sharing requires waiting and overhead, and is a natural enemy of scalability" – Herb Sutter
- The trick to effective concurrency is to do as little synchronization as possible, but no less.

Concurrency in Java

- Divide and Conquer:
  - For a large problem, we want to have at least as many threads as the number of available cores.

- Use Runtime.getRuntime().availableProcessors() to get the number of available cores;

  - Number of threads = Number of Available Cores / (1 - Blocking Coefficient) where blocking coefficient is between 0 and 1.

  - A computation intensive task has a blocking coefficient of 0 and an I/O intensive task has a value close to 1.

Threads: Thread Life Cycle

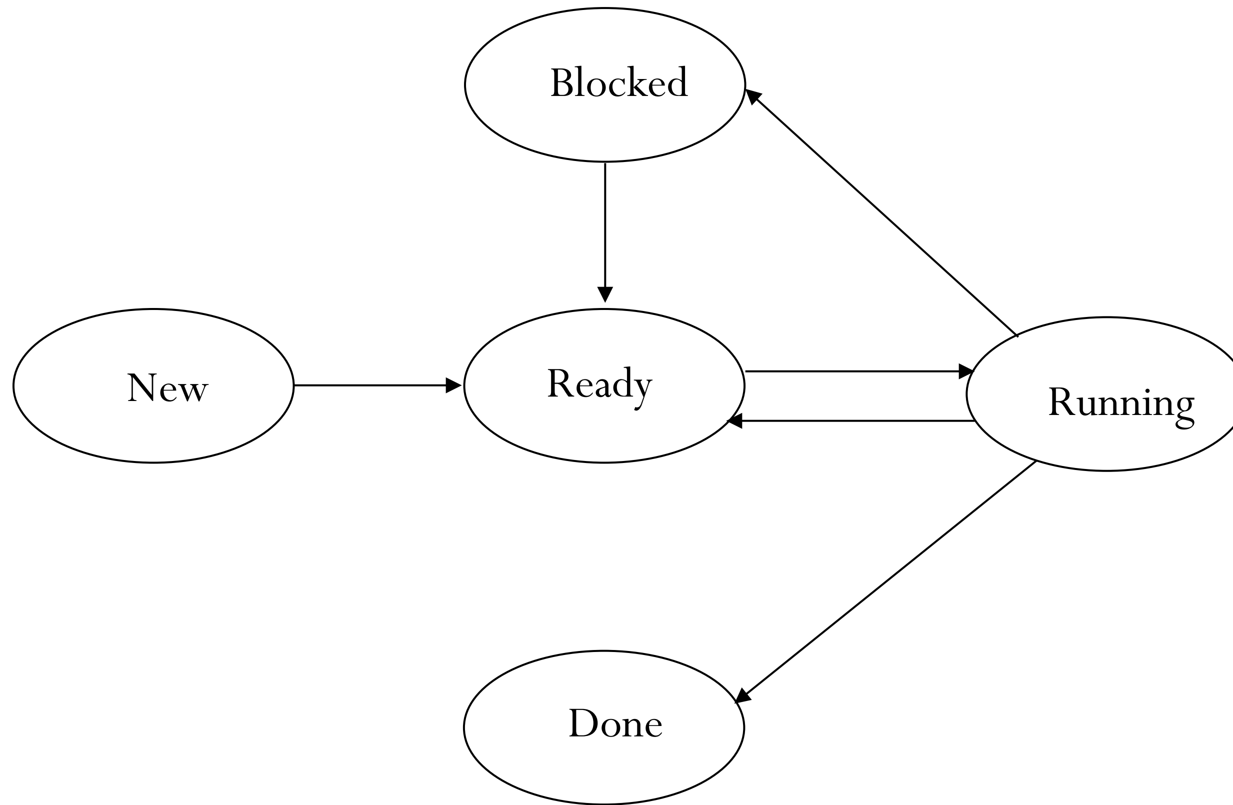


Figure 1: Thread Life Cycle and Scheduler Context switches the thread among these states.

Java Threading API deficiencies

- Not Efficient and Not Scalable: It will use and throw away the instances of the Thread class since they don't allow you to restart.
  - To handle multiple tasks you typically create multiple threads instead of reusing them.

- Hard Synchronization Problem:
  - Methods like wait ( ) and notify ( ) require synchronization and are quite hard to to get right when used to communicate between threads. The synchronized keyword lacks granularity. It doesn't give you a way to timeout if you did not gain the lock. get right when used to communicate between threads.

  - Other Concurrency Issues:
    - Race Condition, Non-Atomic Access to shared data, The lock is a counter: one thread may lock an object more than once, Deadlock.

Newer Generation of concurrency APIs

- "java.util.concurrent package"
- This package has nicely replaced the old threading API.
  - Wherever you use Thread class and its methods, you can now rely upon the ExecutorService and related classes.
  - Instead of using the synchronized construct, you can rely upon the Lock interface and its methods that give you better control over acquiring locks.
  - Wherever you use wait/notify, you can now use synchronizers like CyclicBarrier and CountdownLatch .
  - Modern Java/JDK Concurrency helps in Scalability and Thread Safety

Brief History of Scala

- Scala is an attempt to integrate features of object-oriented and functional programming into the same language. Created at the Cole Polytechnique Fdrale de Lausanne (EPFL) in 2001 by Martin Odersky (creator of Pizza, GJ and a major contributor to Java generics).

- Actor Model is a mathematical model for concurrent computation introduced in 1973 by Carl Hewitt, Peter Bishop, and Richard Steiger.

Actor Based Concurrency Model

- Scala stands for **Scalable**, implements the actor model into a main-stream, Object-Oriented Language.
  - The main strength of Scala is the full interoperability with the Java language. Scala programs use Java legacy objects and Scala code is compiled into byte-code the Java Virtual Machine can execute.
  - Full Support to underlying libraries (java.util.concurrent) and support traditional java concurrency semantics.
  - A layer of abstraction on those basic mechanisms and the actors library.
- An actor is a computational entity that, in response to a message it receives, can concurrently.
  - Send a finite number of messages to other actors;
  - Create a finite number of new actors or
  - Designate the behavior to be used for the next message it receives.

Lifecycle of an Actor.

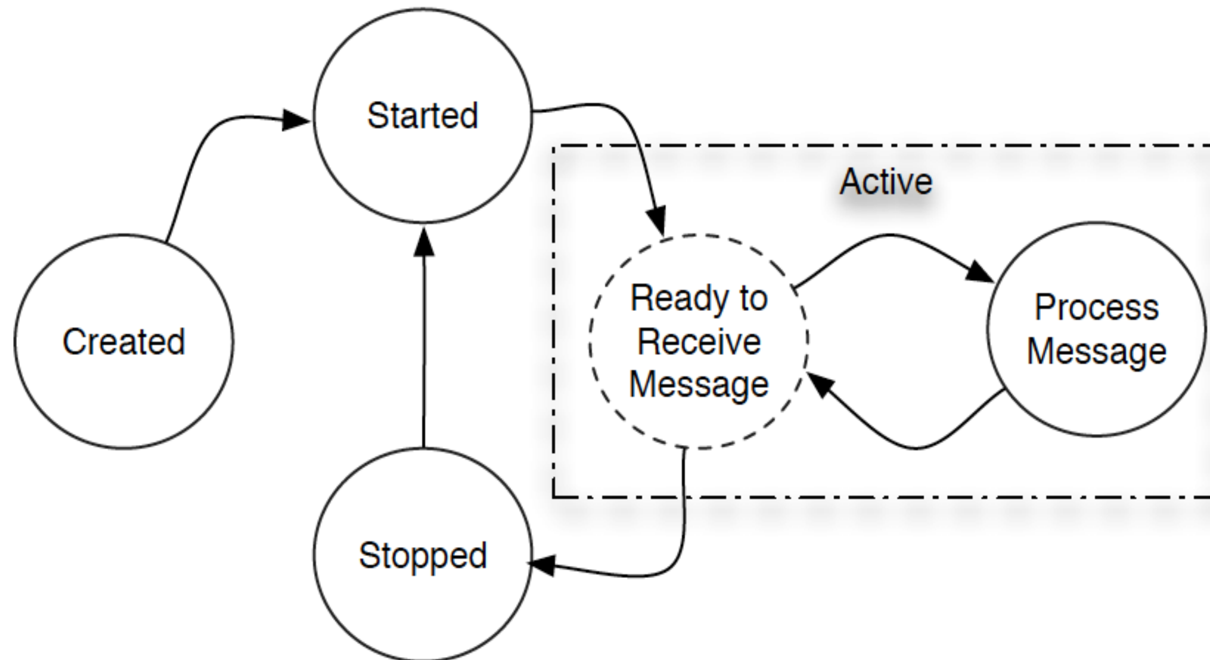


Figure 3: Life Cycle Of an Actor

Actors in Scala are available through the scala.actors library.

Main Points of the Actor Model

- The Actor model is an inherently concurrent model of programming.
- Systems comprise of concurrent, autonomous entities, called actors, and messages.
- Each actor has a unique, immutable name which is required to send a message to that actor.
- An actor name cannot be guessed but may be communicated to other actors.
- Each actor has its own mutable local state; actors do not share this local state with other actors–each actor is responsible for updating its own local state.

Overview of the Actor Model

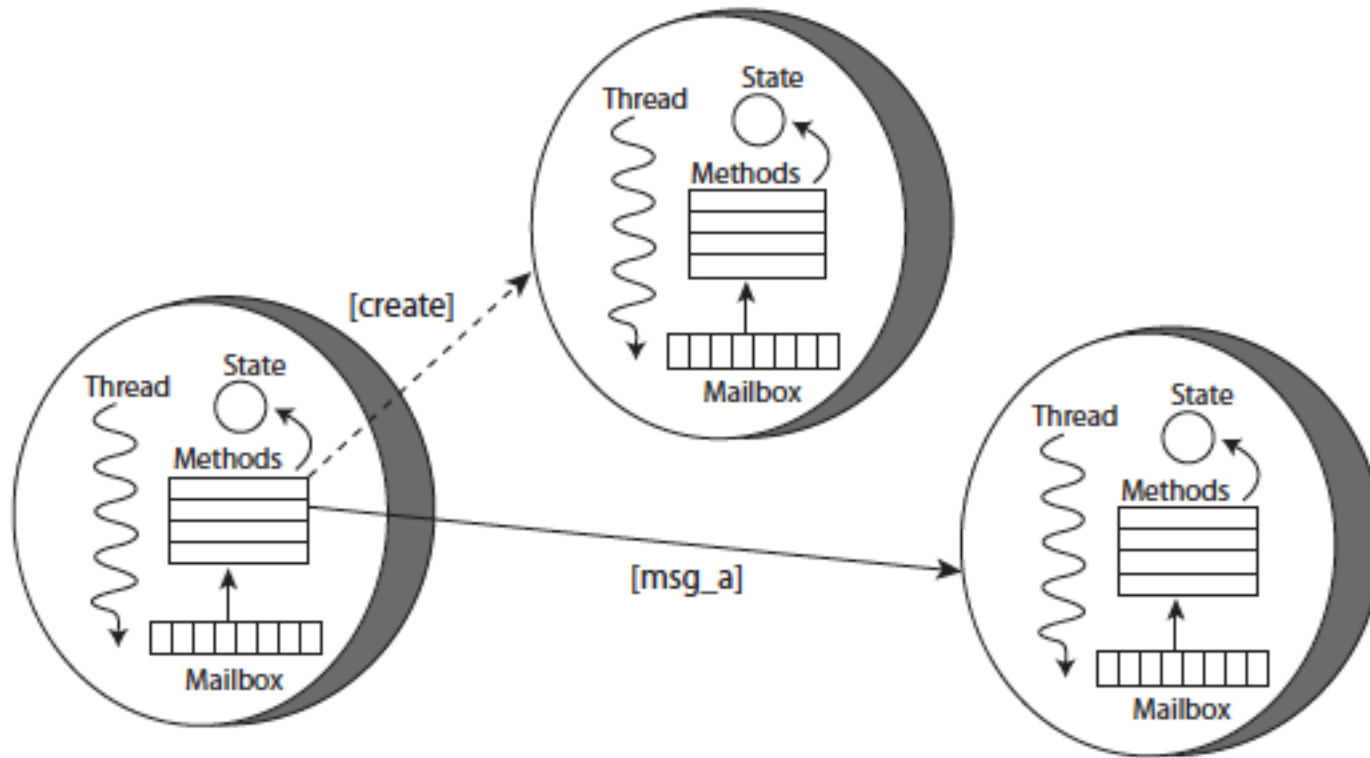


Figure 2: Actors are concurrent entities that exchange messages asynchronously.

Scala compared to Java.

Scala adds	Scala removes
+ a pure object system	- static members
+ operator overloading	- primitive types
+ closures	- break, continue
+ mixin composition with traits	- special treatment of interfaces
+ existential types	- wildcards
+ abstract types	- raw types
+ pattern matching	- enums

Communication Model in Scala

- Communications among actors occur asynchronously using message passing.
  - Recipients of messages (a.k.a. "mailboxes") are identified by addresses, sometimes called "mailing address".

- An actor can only communicate with actors whose addresses it has:
  - It can obtain those from a message it receives, or if the address is for an actor it just created. The actor model gives no guarantee about message ordering, and also buffering is not necessary.

- The strength of this model comes with the use of **mailboxes** which are free of race conditions by definition (messages cannot be read in a inconsistent state).

Scala Actor Properties

- Fairness in Scheduling.
  - A message is eventually delivered to its destination actor, unless the destination actor is permanently "disabled" (in an infinite loop or trying to do an illegal operation).
  - Another notion of fairness states that no actor can be permanently starved.
- State Encapsulation.
  - An actor cannot directly (i.e., in its own stack) access the internal state of another actor. An actor may affect the state of another actor only by sending the second actor a message.
- Safe Messaging.
  - There is no shared state between actors. Message passing should have call-by-value semantics. This may require making a copy of the message contents, even on shared memory platforms.

Sample Code for Fair Scheduling in Scala

```
import scala.actors.Actor
import scala.actors.Actor._
object fairness {
class FairActor() extends Actor {
...
def act() { loop { react {
case (v : int) => {
data = v  }
case (”wait”) => {
// busy-waiting section
if (data > 0) println(data)
else self ! ”wait”  }
case (”start”) => {
calc ! (”add”, 4, 5)
self ! ”wait”
} } }
 } }
 }
```

A program written in the Scala Actors showing an Actor “busy-waiting” for a reply. In the absence of fair scheduling, such an actor can potentially starve other actors.

Scala Actor Properties – Contd.

- Location Transparency:
  - Location transparency provides an infrastructure for programmers so that they can program without worrying about the actual physical locations.
  - Because one actor does not know the address space of another actor, a desirable consequence of location transparency is state encapsulation.
  - Scala Actors, an actor's name is a memory reference, respectively, to the object representation of the actors (Scala).

- Mobility:
  - Location transparent naming also facilitates runtime migration of actors to different nodes, or mobility. Migration can enable runtime optimizations for load-balancing and fault-tolerance.
  - Actors provide modularity of control and encapsulation, mobility is quite natural to the Actor model.

Isolating Mutability using Actors:

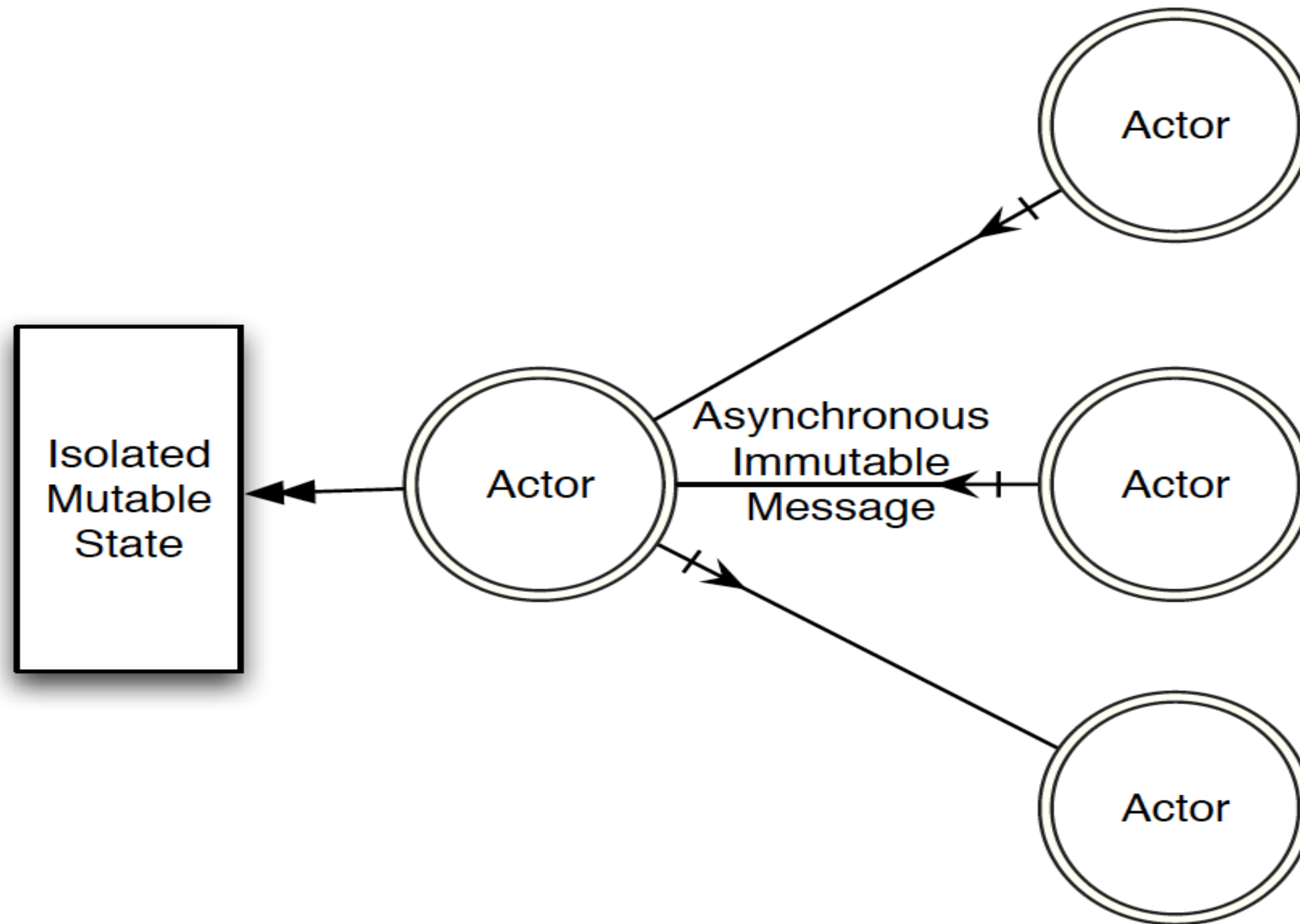


Figure 3: Actors isolate mutable state and communicate by passing immutable messages.

Actor-based Mutability

- Isolating Mutability using Actors:
  - Shared mutability: where multiple threads can modify a variable is the root of concurrency problems.
  - Isolated mutability: where only one thread (or actor) can access a mutable variable, ever - is a nice compromise that removes most concurrency concerns.
- While your application is multithreaded, the actors themselves are single light weight threaded.
- There is no visibility and race condition concerns.
- Actors request operations to be performed, but they don't reach over the mutable state managed by other actors.

Programming Abstractions

- Two useful programming abstractions for communication and synchronization in actor programs:
  - Request-Reply Messaging Pattern
  - Local Synchronization Constraints

Request-Reply Messaging Pattern

- In this pattern, the sender of a message blocks waiting for the reply to arrive before it can proceed. This RPC-like pattern is sometimes also called synchronous messaging. For example, an actor that requests a stock quote from a broker needs to wait for the quote to arrive before it can make a decision whether or not to buy the stock.

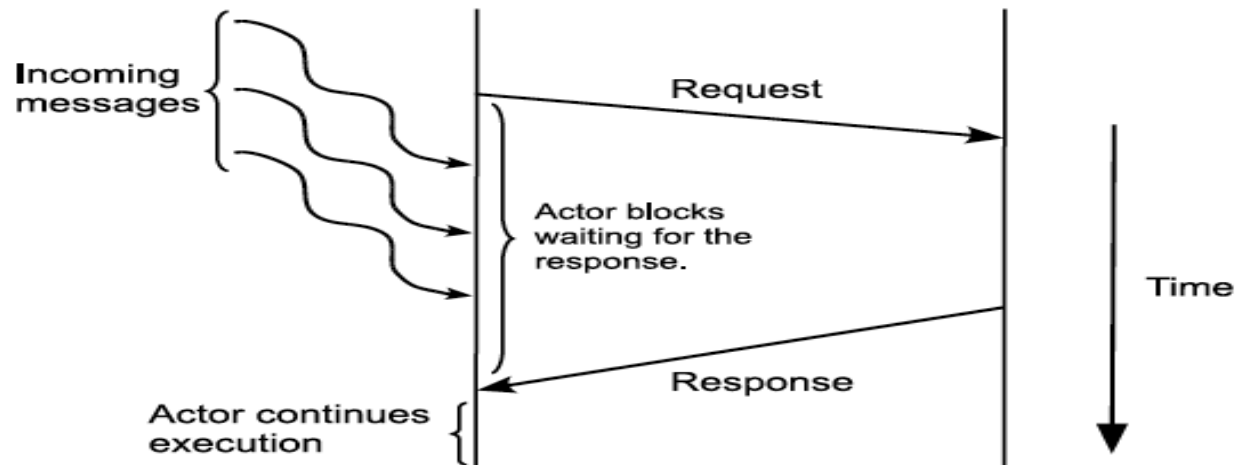


Figure 4 : Request-reply messaging pattern blocks the sender of a request until it receives the reply. All other incoming messages during this period are deferred for later processing.

- Request-reply messaging is almost universally supported in Actor languages and frameworks. It is available as a primitive in Scala Actors.

Scala Actors Local Synchronization

- Scala Actors library provides local synchronization constraints.
- Its support for constraints is based on pattern matching on incoming messages. Messages that do not match the receive pattern are postponed and may be matched by a different pattern later in execution.

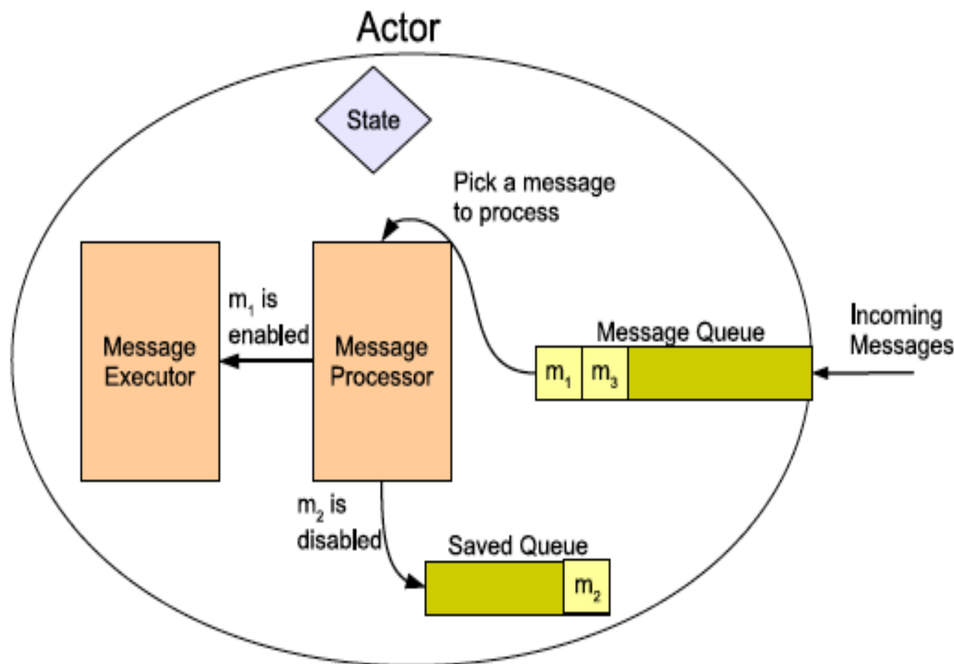


Figure 5: Implementation semantics of local synchronization Constraints

At runtime, a message is processed if it is not disabled i.e., no constraint returns true for the message (see Figure 5). A disabled message is placed in a queue called save queue for later processing. Whenever a message is processed successfully by an actor, it is possible that a previously disabled message (a message in save queue) is no longer disabled. This is because the state of the actor may change after processing a message, and this change of state may enable other messages.

Local Synchronization Constraints

- Each actor operates asynchronously and message passing is also subject to arbitrary communication delays; therefore, the order of messages processed by an actor is nondeterministic.
- Sometimes an actor needs to process messages in a specific order. This requires that the order in which messages are processed is restricted.
- Synchronization constraints simplify the task of programming such restrictions on the order in which messages are processed.

Concurrent Programming in Scala - I

- Scala provides two kinds of concurrent programming models:
  - **Shared Memory** - Provided by means of threads.
  - **Distributed Memory** - Asynchronous message passing provided by the actor model.

- The Scala Actors library defines the actor type and three operators:
  - **a !? msg** (the send operator): sends **msg** to **a**, waits for a reply and returns it.
  - **Receive** and
  - **React**.

Concurrent Programming in Scala - II

- Scala Actors unify both programming models:
  - **Thread-based** or **Event-based**. Developers can trade efficiency for flexibility in a fine-grained way.

- **Thread-based implementation**: Each Actor is executed by a thread. The execution state is maintained by an associated thread stack.

- **Event-based implementation**: An Actor behavior is defined by a set of event handlers which are called from inside an event loop. The execution state of a concurrent process is maintained by an associated record of object.

Threading Performance Evaluation

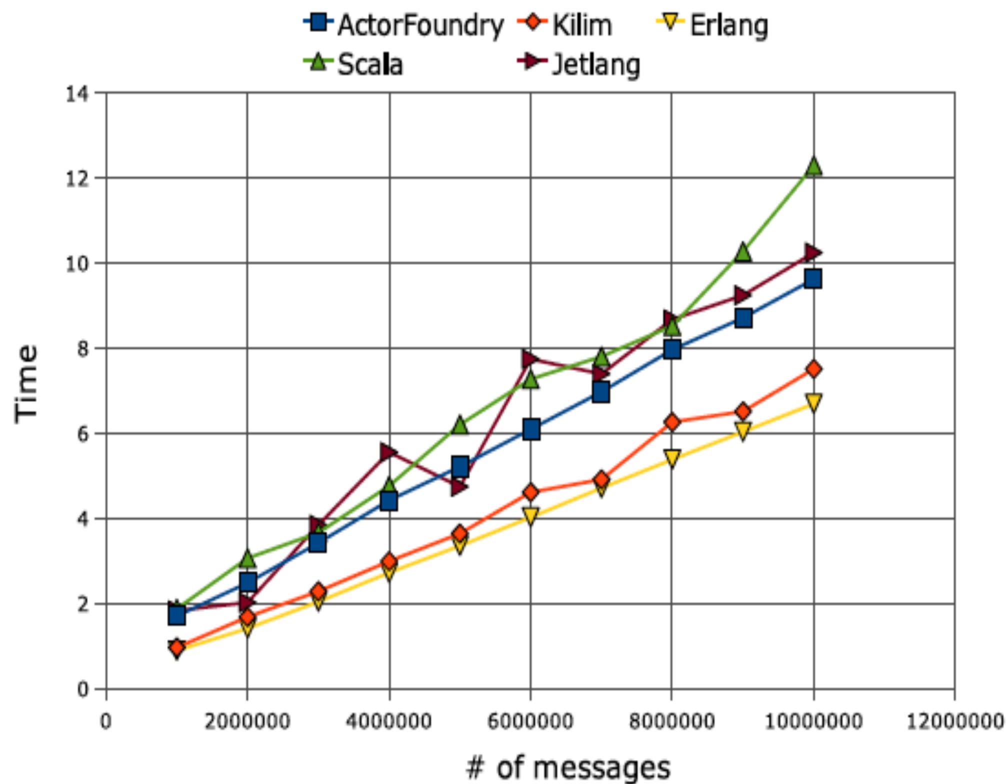


Figure 6: Threading Performance with other Frameworks and Scala.

Observe that Kilim outperforms the rest (including Scala), since the framework provides light-weight actors and basic message passing support only.

The programming model is low-level as the programmer has to directly deal with mailboxes, and it does not provide standard Actor semantics and common programming abstractions. This allows Kilim to avoid the costs associated with providing these features.

ActorFoundry's performance is quite comparable to the other frameworks. This is despite the fact that ActorFoundry v1.0 preserves encapsulation, fairness, locationtransparency and mobility as in Scala.

Performance comparisons of Scala

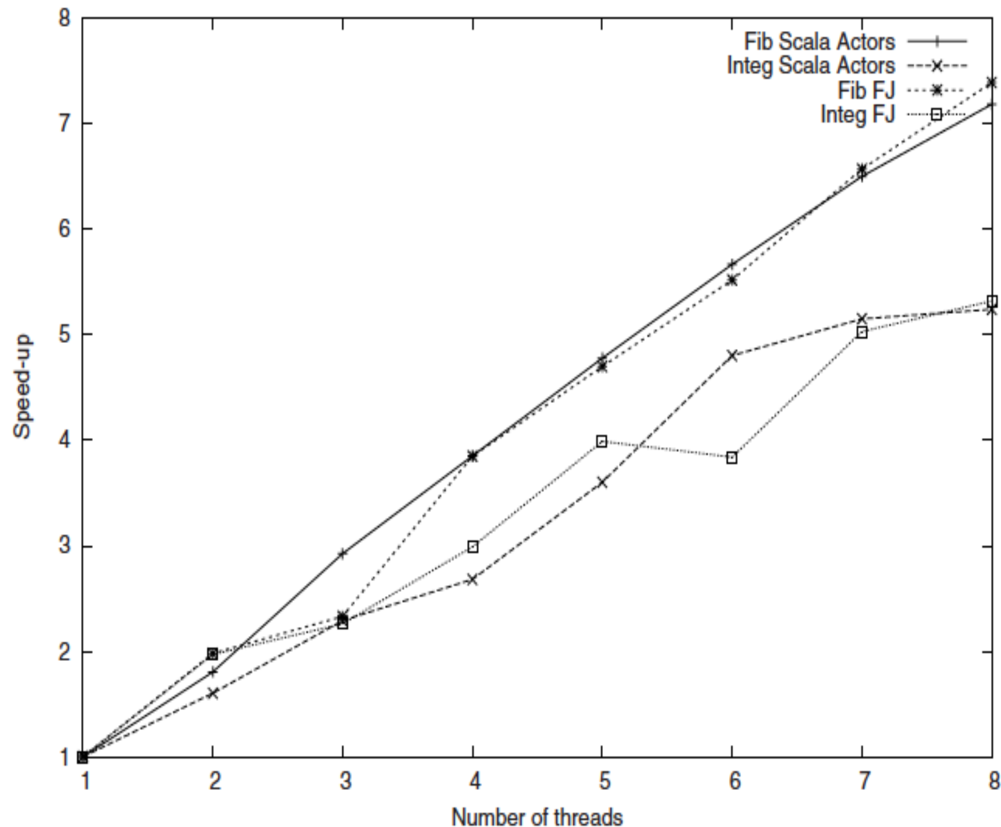


Figure 7: Speed up for Fibonacci Series with number of threads.

- The speed-ups as shown in Figure 7 are linear as expected since the programs run almost entirely in parallel.

Conclusions

- Scala offers modern perspective to Object Oriented Language Design.
  - Scala exploits traits to make objects simpler and easier to build.
  - The functional nature of core syntactical elements provide useful features.
  - Scala provides easier coding for concurrency and writing thread-safe code.
  - Statically typed language.
  - High performance.
  - High interoperability with Java.
  - Support for modularity and extensibility and Scalability.
  - Built-in XML support

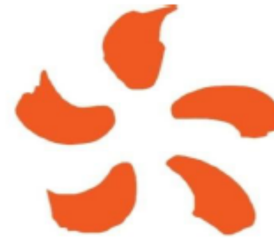
- Despite a growing interest in the Actor model, the model may not be generally well understood beyond the basic concept of actors and asynchronous messages.

- However it is hard for programmers, and the designers of an Actor framework, to understand the implications of the various design decisions in building or using a Scala Framework.

- Results suggest that safe messaging is the dominant source of inefficiency in actor systems.

Scala Users:

Xebia



GridGain
CLOUD COMPUTING

LinkedIn

eDF

imageworks

Who is using Scala ?

SIEMENS

foursquare

twitter

nature

UNIBET

Q & A

Thank You!