Theory and Application of Multithreading

The Actor Model

H.M.F. Lohstroh
marten@stdio.nl

June 25, 2011

Abstract

Computer science arrived at an age where concurrency is more prevalent than it ever was before. Yet much of the reasoning about computation that is reflected by imperative programming languages, is deeply rooted by ideas that find their origin in an age where computation was defined to be sequential. Instead, super-Turing models like the Actor model allow for true concurrency oriented programming languages like Erlang. This paper describes what defines the Actor model and explores the capabilities of Erlang in the context of multithreading.

1 Introduction

The advent of modern computer science was heralded by Allen Turing in 1936 when he raised the first generic theoretical model of computation [15]. The elegantly simple programmable machine that Turing described was a pure sequential one and it made computer programs to be conceived as sequential orderings of instructions to be executed by a single machine. Around the time Turing invented his abstract machine model, Alonzo Church developed his $\lambda$-calculus [4], which comprises a formal system for function definition, function application and recursion. Surprisingly enough the two models are essentially equivalent in the sense that they are encodable into each other. But yet again, both are models for pure sequential computation.

During the early 1940s the first electro-mechanical and electronic computers emerged and initially these machines indeed operated strictly sequentially, but in the quest for performance gain, from the late 1950s computer architects started to incorporate parallelism in their designs. Since that time the landscape of computing changed a lot. During the early 1970s the first microprocessors became commercially available after which ongoing miniaturization and clock frequency scaling allowed for processors to become increasingly more complex and fast. With the reduced size and increased power of computers, more and more applications became eligible to be automated. Hence computer applications too became significantly more complex. With the increasingly wider range of applications, computers became a lot cheaper and more common. Meanwhile, computer networks became ubiquitous and developed into a global
infrastructure allowing computers to interconnect and cooperate from virtually anywhere. These days computers are truly omnipresent, typically carrying multiple processors and being connected to some sort of network.

So we went from simple to complex, from sequential to parallel, and from isolated to connected. Taking the notions parallel and connected together we arrive at the notion of concurrency, which is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other. Concurrency imposes synchronization problems upon systems that if not managed well can exhibit undesired behavior such as deadlock, livelock, or starvation. Complexity makes understanding concurrent systems a problem. Meanwhile we rely on these systems to fly our planes and manage our stock exchanges, so they need to be reliable. Assuring that concurrent systems indeed behave as intended, calls for a thorough understanding of their working, yet we still need to map our comprehensible algorithms onto complex hardware, where much of this understanding is lost in translation.

Testing on the level of implementation is by no means an assurance that undesirable behavior is completely ruled out. A formalism that captures concurrency in a comprehensible way, can help understand the intricacies of concurrent programming while it opens the gate toward model-checking capabilities. Many efforts have been made to raise a formal model of concurrency; the Actor model that was introduced by Hewitt, Bishop and Steiger [9] in 1973 is one of them. This particular model has gained renewed interest in recent years and has seen several successful implementations, of which most notably Erlang [2]. This paper describes what defines the Actor model and explores the capabilities of Erlang in the context of multithreading.

2 Concurrency

The basic unit of concurrency is a process that performs a series of computational steps. In order to produce meaningful results, processes typically operate in a deterministic fashion, meaning that for each computational step the following step is predetermined. This is exactly the operation that a classical Turing machine describes and can be represented using a deterministic finite automaton. Concurrency is a property of systems in which several (deterministic) processes execute simultaneously. It is possible that these processes interact, which gives rise to nondeterminism, meaning it is no longer clear in what state a process might end up. This may result in incorrect computation due to race conditions, unexpected failure due to deadlock or livelock, or starvation as a result of perpetually being denied the required resources to finish a task.

The technical difference between a process and a thread is only relevant in the context of a shared memory system, where processes operate in an isolated context whereas threads can operate in a shared context. The key observation of the basic unit of concurrency is that it is again sequential, indeed without lodging any form of parallelism inside. Processes operate autonomously within a system that accommodates for resources. Through a system, processes exhibit behavior by executing instructions that affect the state of the system. When processes run concurrently, the
behavior of the system — characterized by the observed state changes — is determined by the interleaving of executed instructions. While the latter is described as “apparent” or “pseudo” concurrency, the use of multiple processing units allows for “real” concurrency where computational steps are carried out truly in parallel.

To understand behavior, it does not matter whether concurrent processes reside within a shared memory system, or whether they are distributed over multiple machines. Interaction can either take place explicitly through signals or messages, or implicitly by means of shared access to mutable resources. Although message passing seems less fitted for inter processes communication as shared memory takes away the need for messaging, the message passing paradigm is certainly more general. In fact, message passing can be implemented through shared memory (and vice versa), but it does not rely on the assumption that there is such a thing as shared memory. Moreover, communication (or sharing, if you will) is made explicit using message passing semantics whereas communication through shared memory is implicit and therefore arguably less suitable to comprehensively describe the workings of a system. The use of shared memory is better thought of as a computer architectural optimization that implements the fastest possible inter process communication. Whether this capability is exploited, should be regarded a matter of implementation. Just as well, a system could be implemented using multiple machines, jointly constituting a distributed system. Message passing semantics offer the advantage that even when threads migrate from one machine to another, the operational semantics do not change. Not surprisingly, most existing models of concurrent computation embrace the message passing paradigm to describe communication.

In the same sense that message passing is more general than communication through shared memory, asynchronous communication is more general than synchronous communication. Using a two-phase commit protocol, synchronized communication can be modeled using asynchronous communication. However, asynchronous communication cannot be implemented by a Turing Machine, because the reception of messages cannot be logically inferred [8]. Hereby — with a model that offers asynchronous message passing semantics — we arrive at the notion of hypercomputation, or super-Turing computation. So in order to provide the most generic model of concurrent computation, the Turing machine falls short.

Concurrent execution implies the sharing of resources. To control access of those resources, concurrent systems require the inclusion of some kind of arbiter. To give an example, consider an interrupt handler that suspends the currently executing process, allowing another process to continue. Such arbitrage gives rise to indeterminacy in concurrent computation. Turings model for sequential computation, as well as λ-calculus and the later invented finite state machines, make use of a global state to represent a computational step. This makes that even nondeterministic versions of these models, have the property of bounded nondeterminism, which was proven informally by Gordon Plotkin in 1976 [14]. It means that if a machine always halts when started in its initial state, then there is a bound on the number of states in which it can halt, i.e., it cannot go through an unbounded number of states before it halts.
As stated in [8], a system that exhibits unbounded nondeterminism can be described using the following two rules:

1. When started, the system always halts.
2. For every integer \( n \), the system can halt with an output that is greater than \( n \).

Unbound nondeterminism (also called unbound indeterminacy) allows the amount of delay in servicing a request to become unbounded as a result of arbitration of contention for shared resources, while still guaranteeing that the request will eventually be serviced. In a client-server setting, this is a realistic situation as a request to a shared resource might never receive service because it is possible that a nondeterministic choice will always be made to service another request instead. Hence, semantics of unbounded nondeterminism are required to prove that a server provides service to every client. As suggested in [5], super-Turing models that feature asynchrony and unbounded nondeterminism can be expected to become a central paradigm of computer science. As a side note, this would raise interesting questions in the field of computational complexity theory, that is of course pinned onto the concept of algorithmic computation as described by Turing machines.

Several formalisms for modeling and understanding concurrent systems have been developed throughout the past few decades. Worth mentioning are process calculi of which most find their origin in CCS [12], CSP [10], or ACP [3]. Process calculi model concurrent systems through transition systems that feature algebraic laws that permit reasoning about equivalences between processes in terms of bisimulation.

Petri nets [13] are also strongly represented in the literature, mostly in the field of reasoning about distributed systems. A petri net is a directed bipartite graph that describes relations between nodes that either represent transitions or places. Tokens travel through the graph by means of the nondeterministic firing of transitions that transfer tokens from their input places to their output places.

Also noteworthy are temporal logics such as HML [7] and TLA [11] that find their principal application in writing specifications for concurrent systems. Many more models exists, but since the focus of this paper lies with the Actor model, no attempt has been made to summarize the entire landscape of formalisms.

3 The Actor model

The Actor model is a model of concurrent computation that evolves around the notion of “Actors”. The model finds its foundations in physics instead of mathematics or logic. Similar to the philosophy of object-orientated programming languages where everything is an object that provides methods and maintains some state, the Actor model reasons about entities in terms of stateful actors that can exchange messages.

In response to a message it receives, an actor can concurrently:

- send a finite number of messages to other actors;
- create a finite number of new actors;
- determine how to handle the next incoming message.
As these actions could be carried out concurrently, there is no assumed order to them. Hence messages that are sent concurrently, arrive in arbitrary order. Perhaps a bit surprisingly, this also applies to messages sent to the same actor. This is due to the fact that sending and receiving is decoupled and thus communication between actors is asynchronous.

Concurrent operation and asynchronous communication makes that there is no restriction on message arrival order at all. This property gives rise to unbound nondeterminism. For example, in the Actor model it is possible for a terminating program to generate an infinite sequence of numbers. Assume that actor A sends a message containing \{stop\} concurrently with a message containing \{start, 1\} to actor B. Upon the receival of \{stop\} actor B shall terminate, whereas upon the receival of \{start, i\} actor B prints \{i\} and sends \{start, i + 1\} to itself. A nondeterministic choice in the receival of messages can make that the \{start, i\} messages always arrive before the initial \{stop\} message is delivered. An illustration is given in Figure 1.

![Figure 1: Computing an integer of unbound size.](image)

The Actor model captures “real” concurrency, but in a practical implementation actors need to be mapped onto processes that might very well be executed in an interleaved fashion. Then again, actors operate inherently concurrent, which is different from e.g. process calculi, wherein concurrency is achieved by means of parallel composition of sequential processes.

Actors are assumed to live in some address space. In contrast with process calculi that allow anonymous processes to communicate through named channels, the Actor model provides communication between actors by means of their addresses. Besides the fact that this construct does not impose any constraints on the underlying communication network, it allows for a dynamic communication topology through the exchange of addresses. Once an actor receives a new address from another actor, it can start sending messages to the actor listening on that address. After all, the Actor model does not put any restriction on the content of messages.

Having that the Actor model features asynchronous message passing which is arguably the most generic way describing communication, it is suitable to model a wide range of concurrent systems like e-mail, webservies, objects with locks, dynamically routed packet-switched networks, and much more. Most importantly, the Actor model abstracts away from implementation details and low level machine properties. There is for example no notion of hardware threads, operating system processes, or the scheduling of them. Although these constructs are real, they belong in the domain of the operating system and underlying hardware. There is abso-
olutely no need for their semantics to reflect in the programming model. As a result, the semantics of a concurrent system described using actors, are truly independent of the underlying machine model. There is however one problem that sticks to such abstraction, particularly in combination with unbound nondeterminism, which is lack of resource awareness. A model that is resource unbounded yet resource agnostic, is unsuitable to prove anything in a real-world situation in which resources are finite. Along its path of execution a program might simply exhaust all resources and come to a grinding hold. However, in such practical setting, the use of resources can typically be found to scale proportionally to some parameter like the size of a given input or the number of users in a system, which makes it fairly predictable. Hence there is no need to prove anything formally; statistics would suffice to ensure proper service. The latter would be an argument against building resource awareness into the model at the cost of making it needlessly complex.

The Actor model has seen many implementations, both natively by full-fledged programming languages, as through add-on libraries that provide actor semantics to existing languages. Examples of early implementations are the languages Act, Cantor, and Rosette, more recent ones are Erlang, Salsa, and Scala. Popular languages like Java, C++, and Python feature several libraries or actor frameworks that permit actor-style programming.

4 Erlang

Erlang is a purely functional language that implements the Actor model\(^1\), thereby raising a programming model that features formal operational semantics. It was originally designed for programming fault tolerant distributed systems at Ericsson and was released into the public domain in 2000. The principal inventor of Erlang, Joe Armstrong, argues that the world we live in is inherently concurrent, thereby making it paradoxical that the program languages that we use to write programs that interact with the world, are predominantly sequential. He blames this state of affairs for the common conception that concurrent programming is difficult and poses that this is not due to the nature of concurrency itself. With Erlang, he raised a programming language in which writing concurrent programs is the default mode of expression. Analogous with object orientation, he coined the term “concurrency oriented programming” \([1]\) to characterize Erlang as a language that has good support for concurrency. The remainder of this section will attempt to explain why.

By implementing the Actor model, Erlang wholly embraces the “shared nothing” concept. Programming multithreaded applications in imperative languages like Java or C++ is intricate and error prone as the programmer is burdened with the task to preserve the integrity of shared state. Access to shared resources can be serialized using locks, but the order in which these locks are obtained and released is crucial in order to prevent deadlock. However, oversynchronization that comes as a result of too coarse grained locking lets applications tend toward single-threading. On the other hand, when locking is too fine grained, the chance of running into deadlock dramatically increases as it becomes much harder to figure

\(^{1}\)Which is rarely commented on in literature about Erlang.
out and enforce the correct order in which locking should take place. To
circumvent this trouble, Erlang simply does not allow shared state. It
strictly follows the message passing paradigm of the Actor model to ex-
change information between Erlang processes. The immutability of vari-
ables makes that protection from concurrent access is no longer needed.
Erlang features pattern matching instead of traditional assignment. The
‘=’-operator yields success when the value of the variable on the left-hand
side matches the value on the right-hand side. Assignment only takes place
when the variable on the left-hand side is unbound. Having immutable
variables makes that wrongly assigned values can be traced down a lot
easier as there can only be a single line of code that is responsible for that
assignment. Clearly, the absence of shared variables allows for a higher
degree of parallelism, which however can only be exploited given that Er-
lang processes are not too heavy-weight.

Kernel threads are too heavy-weight in terms of creation time and
context switching overhead to support massive parallelism, let alone that
they are also quite limited in the number available. Therefore, Erlang
makes use of so called light-weight processes (LWP), of which multiple
run in user space within a single kernel thread, hence sharing a single
address space and system resources. The scheduling of Erlang processes
as LWPs is taken care of by the Erlang execution environment that is im-
plemented through a virtual machine (VM). Like Java or Python, Erlang
uses a stack-based byte-code VM. Scheduling of processes is preemptive,
meaning that execution of a process is temporarily suspended on behalf
of another process upon the reach of a time limit or when it blocks. Since
2006 the Erlang run-time environment has support for symmetric multi-
processing (SMP) i.e., it is able to map LWPs onto kernel threads that
are scheduled to run on different cores. Hence it takes full profit of multi
core architectures.

A major advantage of having a true concurrent model of computation
underlying the programming language, is that concurrency is no longer
a property of the operating system, but a property of the programming
language and the accompanied execution environment. This means that
concurrent behavior of a program is the same on any platform, regard-
less of any underlying scheduling order, issues with synchronization, or the
amount of available resources. The only notable difference between execu-
tion on different systems should be the speed of it. The message paradigm
that naturally extends over networks makes it possible to use single ma-
chine Erlang code without too much effort in a distributed setting. Turn-
ing a non-distributed program into a distributed program is merely a
matter of different resource allocation. Scalability is achieved on the level
of individual systems by the enormous amounts of Erlang processes that
can be maintained without serious performance drawbacks. It was shown
in [1] that a webservice written in Erlang happily manages 85,000 pro-
cesses with only little observable performance degradation, while in the
same experiment, the well-known Apache webserver succumbed under the
load of only 4,000 parallel sessions and crashed. In a distributed setting,
processes can conveniently be mapped onto more processors, which allows
further scaling. It should be noted that despite its generic form, message
passing in Erlang is extremely efficient. Armstrong claimed in [1] that
process creation times and message passing times are one to two orders of
magnitude faster than equivalent operations on threads in Java or C#.
Erlang systems typically comprise large numbers of processes. When having so many processes, not much harm is done when a single process dies. This as opposed to sequential programs that by themselves are single points of failure. Single threaded programs only have a single chance to recover from an error or handle a certain exception, an Erlang system is much more flexible. The basic idea is that processes fall into two categories; workers that carry out some computation, and supervisors that monitor the workers and undertake appropriate action when required. An important argument for this scheme is that local error handling and recovery is simply not always possible, e.g. in the case of hardware failure. Therefore, letting failing processes die while having supervisors to restart them, allows for a very high level of fault tolerance. Another interesting feature that constitutes to the high level of availability that can be achieved in Erlang systems, is hot code replacement. The message passing paradigm in combination with pattern matching allows for the exchange of code segments that define subsequent behavior. So without bringing the entire system to a hold, individual components can be updated, all while continuing service.

Despite the enormously rich feature set that Erlang has to offer, it has difficulty to gain interest from the larger public. This has much to do with the fact that it is a functional language, which by many is considered an oddity in the landscape of programming languages that is still dominated by the imperative programming paradigm. Functional programming requires a different way of reasoning, but there is no compelling evidence of it being a harder one. Strictly sequential programs are more difficult to optimize in Erlang than it is in C or ultimately, in assembly language. This is simply due to the fact that imperative languages are closer related to the step-wise operation of the machine, hence they allow for more detailed description and stricter guidance of the computation. On the other hand, functional languages allow far easier optimization of concurrent programs. In the age when the C programming language was invented, resources were scarce and machines operated predominantly in a sequential fashion. Nowadays, there exists a wealth of resources, and concurrent execution is the only way of exploiting them. Hence, optimization in terms of sequential execution is starting to lose its meaning.

Although functional programming has received a significant amount of attention in academia, it had a hard time finding its way to industrial applications. Erlang was in that respect a real game changer and has seen many successful commercial applications. Erlang was initially developed by Ericsson for the use of building reliable telecom switches, of which the AXD301 that features a reliability of a staggering 99,9999999% is the most striking example. After Erlang was released into the public domain, many other commercial parties adopted it and started to develop their own applications. To give an example, Erlang was used by Facebook to implement the distributed system that supports online chat functionality between its more than 500 million users.

As a final note on Erlang, quite recently a modelchecker for Erlang has come into existence. It is called McErlang [6] and the tool itself is also written in Erlang. It is said to be far from finished, but it has already shown to be able to track down errors in distributed Erlang programs.
5 Conclusions

In order to bridge the gap between concurrency theory and the daily practice of computer programming, the model that is used to reason formally about a program must translate well into the domain of the programming language. Without formal operational semantics of concurrent computation, the behavior of a program is subject to uncontrollable influences caused by constructs that exist on the level of the operating system, or the underlying machine model. The semantics of a program should not depend on the specifics of the hardware that executes them. This opinion is based on the conjecture that it will not be feasible by programmers or compilers to optimize for massive concurrent execution through reasoning in terms of low level machine constructs. The concept of a process serves well as primitive to manage concurrency as it offers a proper level of abstraction. The behavior of a process is well-defined as long as there is no notion of shared mutable resources, hence shared memory should not be part of the programming model. The paradigm of asynchronous message passing as featured by the Actor model, is arguably the most generic. Particularly, the Actor model is more expressive than a Turing machine.

Erlang implements the semantics of the Actor model and manages to abstract away from hardware and operating system details by raising its own well-defined execution environment by means of a virtual machine. By pulling the scheduling of light-weight processes into its own controlled environment and pushing away the management of heavy-weight processes and kernel threads into the domain of the operating system, it can ensure that programs indeed follow the semantics as defined in the programming model.

In the end, we want to describe our programs comprehensively, assure they result in the correct behavior and run them with optimal effectiveness. Whether Erlang is the best language to write comprehensive code is disputable and to a certain extend a matter of taste. However, what Erlang undeniably succeeded in is capturing formal semantics of concurrent computation in the programming language, right where they belong. When it comes to optimization of code, it is starting to matter less whether we can make code run fast on one core, as in reality we typically have multiple at our disposal.
References


