System Architectures
Actors: The Actor Model

Jonathan Thaler

Department of Computer Science

FH Vorarlberg
University of Applied Sciences

Moore's Law

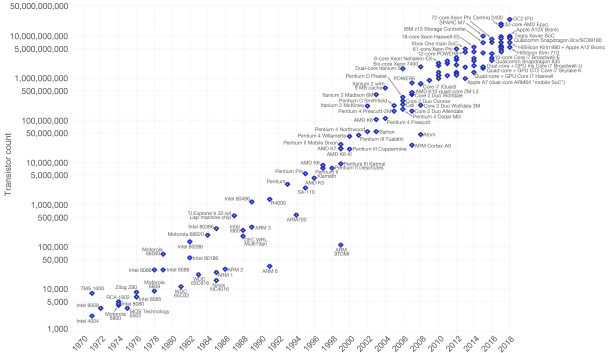


Figure: Moore's Law 1971 - 2018.

The number of transistors doubles about every two years.

This does **NOT** say nor mean that the CPU speed/clock doubles about every two years!

Introduction

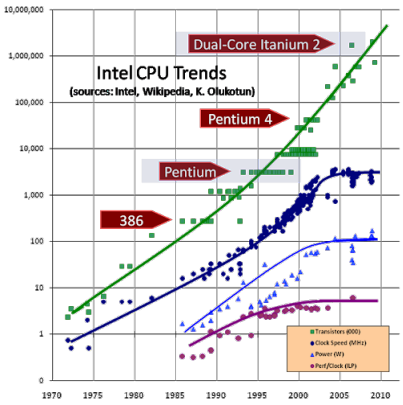


Figure: *The free lunch is over!* (Herb Sutter, 2005)

As of 2005...

- CPU serial-processing speed is reaching a **physical limit**.
- CPU manufacturers will focus on products that better support multithreading such as **multi-core** CPUs.
- Software developers are forced to **adopt** their programs to make use of multi core CPUs by exploiting parallelism and concurrency through **multithreading**.

How do we deal with this challenge?

Introduction

Shared Mutable State

- Multiple processes or threads operate on **shared state**, which they can arbitrarily **read** and **write**.
- The challenge is to get the **synchronisation** right: whenever mutable shared state is modified, it must not be concurrently accessed.
- Various **synchronisation techniques** exist such as monitors, semaphors, mutexes, read-write locks, software-transactional memory,...
- Various **patterns** for various parallel and concurrent problems have been found, such as **producer-consumer**.

Introduction

Message Passing Concurrency

- The problem is split into multiple **autonomous processes** (threads) which have their own local memory and communicate with each other through **messages**.
- The messages contain only **immutable** data and **no references** to local memory of a process / thread.
- **No information is shared** between processes / threads, therefore no concurrent modification of state can happen.
- This is also known as **shared-nothing message-passing concurrency**.

Introduction

We need to embrace concurrency:

- Concurrency has become **increasingly relevant** in the last decade due to the advent and increasing popularity of **multi-core** CPUs.
- Shared-nothing message-passing concurrency is considered to be **easier** to implement and handle than lock-based concurrency building on shared mutable state.
- An instance of message-passing concurrency is the **Actor Model**.
- The **essence** of the Actor Model can be understood to be a mathematical model of **concurrent computation** with **shared-nothing message-passing semantics**.
- **Multiple** software libraries in the field of web applications, concurrent and distributed computation, have built on implementations of the Actor Model such as **Akka**.

Introduction

This course is going to discuss the **conceptual** *Actor Model* as well as a **concrete** Actor Model implementation in the instance of the *Akka* toolkit.

The Actor Model

The Actor Model can be understood to be a mathematical model of concurrent computation with shared-nothing message-passing semantics.

Actor Model

Actors

Actors are **computational agents** which map each incoming communication to a 3-tuple consisting of:

1. A finite set of **communications** (messages) sent to other actors.
2. A new **behaviour**, which will govern the response to the next communication processed.
3. A finite set of **new actors** created.

E-Mail can be modeled as an actor system, where the mail accounts are modeled as actors and email addresses as actor addresses. **Web services** can be modeled with endpoints modeled as actor addresses. **Objects** (with locks, such as in Java/C#) can be modeled as actors.

Actor Model

1. The behaviour of an actor can be **history sensitive**, depending on previously received message, having changed the current behaviour and state.
2. There is **no presumed sequentiality** in the actions an actor performs: mathematically each of of its actions is a function of the actor's behaviour and the incoming communication.
3. Actor creation is an **integral part** of the computational model. In the context of parallel systems, the degree to which a computation can be distributed over its lifetime is an important consideration. **Creation of new actors provides the ability to abstractly increase the distributivity of the computation as it evolves.**

Actor Model

The actor model was inspired by physics:

- Information in each actor is **localised** within that actor and must be **communicated** before it is known to any other agent.
- As long as one assumes that there are limits as to how fast information may travel from one actor to another, the local states of one agent as recorded by another agent relative to the second agent's local states **will be different** from the observations done the other way round.
- The kind of **information** about the physical world that is available to us is the potential consequences of our **experimental interventions** into nature. Therefore, the information about an actor system that is available to us is through consequences of our interventions with the respective actor system.

Actor Model

The way distinct physical systems (actors) affect each other when they **interact**, exhausts all that can be said about the physical world.

The physical world (an actor sytem) is thus seen as a net of interacting components (actors), where there is **no meaning to the state of an isolated system**.

A physical system (an actor system) is **reduced** to the net of relations it entertains with the surrounding systems, and the physical structure of the world is identified as this net of **relationships**.

Actor Model

This has the following consequences:

- A realistic model must assume that the arrival order of communications sent is both **arbitrary** and **entirely unknown**. Therefore, one **cannot** predict when a message sent by one actor will arrive at another.
- Attempting to **observe** the **arrival** of messages affects the results and can even **push** the **nondeterminism elsewhere**. Instead of observing the internals of arbitration processes of actor computations, **we await outcomes**.
- We are foced to **give up** the concept of a description of a system independent from the observer. That is, there is no observer-independent data at all: no concept of **unique (linear) global time** nor an **absolute system state, time and view**.
- This does not mean that there are no relations whatsoever between views of different observers. **It is possible to compare different views, but the process of comparison is always a (phyiscal) interaction (messages)**.

Actor Model

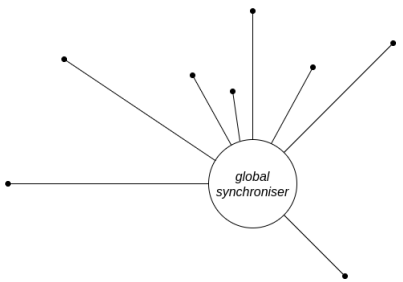


Figure: A global synchronising mechanism.

- A **globally synchronising** mechanism could control the cycles of computation by each of the actors in the system.
- Each actor carries out **one step at a time** and communicates with the global synchroniser for **permission** to carry out the next step.
- Such a mechanism is **extremely inefficient** however, as very actor must wait for the slowest actor to complete its cycle.

Actor Model

An Actor processes always one message at a time - there is no parallel processing of messages within an actor.

- This is a fundamentally **different** concept than concurrency in objects: actors are truly *autonomous*, *active* and *proactive* and can actually **refuse** to process (ignore) messages.
- Methods called on objects (events sent to objects) can come from **arbitrary** sources if they hold a **reference** to that object.
- When multiple **concurrent** sources (threads) are calling into an object, it will then process **multiple messages at the same time**, which results in **desaster** if no synchronisation is happening, in case of **mutable** data.

Actor Model

Futures

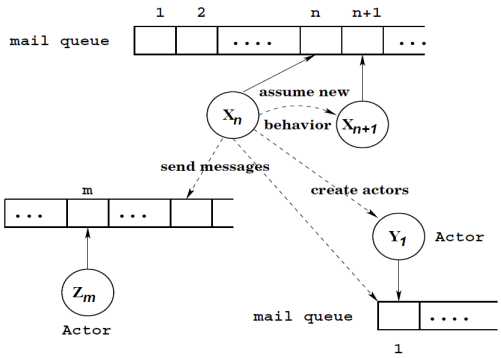
A Future is a **representation** of a (very) long running computation which is created **now** but which can be passed around while it is **computing concurrently**.

- The idea behind a Future is that you can **create** an actor which **eventually** will produce a **result**.
- Eventually you will either **wait** blocking for the result or get **notified** if it has finished.
- Futures can be passed around, stored, sent in messages, all **while** the actual computation is carried out concurrently in an actor specifically created for it.
- Futures **decouple** the producer from the consumer, which is a very powerful concept.
- Using Futures allows actors to send messages to **themselves**, allowing them to be **proactive** and initiate actions by themselves.

Defining an Actor System

Computation in a system of actors is carried out in response to communications sent to the system. Communications are contained in tasks. The configuration of an actor system is defined by the actors it contains as well as the set of unprocessed tasks.

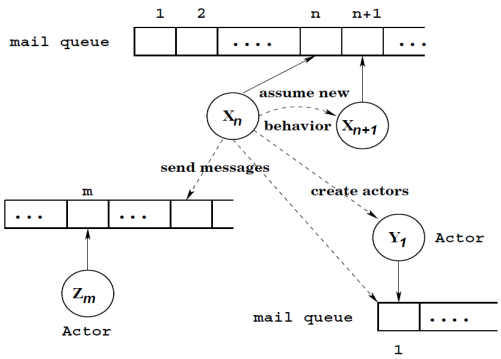
Defining an Actor System



Before an Actor A can communicate with another actor a reaction to some incoming message M, it must **know** the target actor's mail address. There are **three** ways for that:

1. The target was **known** to the actor A **before** it accepted the message M.
2. The target became known when A accepted the **message** M which **contained** the target.
3. The target is the mail address of a **new actor created** as a result of accepting the message M.

Defining an Actor System



- Synchronisation in the actor model is **implicitly** built into the axiom that o**nly one message is handled at a time**.
- The actor is therefore **waiting** for the next message to arrive, which creates a **synchronisation point**: actor X can send a message to actor Z, which is waiting. Actor X then in turn waits for the next message, which actor Z can send.

Defining an Actor System

Programming an Actor System

A program in an actor language consists of:

1. **Behaviour definitions** which simply associate a behaviour schema with an identifier.
2. **New expressions** which create actors.
3. **Send commands** which create tasks.
4. **Receptionist** declaration which lists actors that may receive communications from the outside.
5. **External declaration** which lists actors that are not part of the population defined by the program but to whom communications may be sent from within the configuration.

Defining an Actor System

Defining Behaviours

Each time an actor **accepts** a communication, it computes a **replacement** behaviour.

Since each replacement behaviour will also have a replacement behaviour, and so on, resulting in a potentially infinite definition, we employ **recursive definitions**.

For example, the behaviour of a bank-account depends on the balance in the account. We therefore specify the behaviour of every account as a function of the balance.

Defining an Actor System

Creating Actors

An actor is created using a **new expression** which returns the **mail address** of a newly created actor.

The mail address returned should be **bound** to a variable or used directly in message passing, otherwise it would not have been useful to have created the actor.

Defining an Actor System

Sending Messages

A message is created by specifying the **content** and a **target**.

The target is the **mail address** of the actor to which this message is sent and which might **process** this message.

Messages may be sent to actors that already exist, or to actors that have been newly created by the sender.

Defining an Actor System

Declaring Receptionists

Receptionists are declared, which are actors that are free to **receive communications from outside the system**.

Since actor systems are dynamically evolving and open in nature, the set of receptionists may also be **constantly changing**.

Whenever a communication containing a mail address is sent to an actor outside the system, the actor residing at that mail address can receive communications from the outside, and therefore b**ecome a receptionist**. Thus the set of receptionists may increase as the **system evolves**.

Defining an Actor System

Declaring External Actors

When an actor system is being defined, it maybe intended that it be part of a **larger system** composed of independently developed modules.

Therefore, we allow the ability to declare a sequence of identifiers as **external**.

Such external actors are implemented as **futures**: the identity of such actors is determined sometime after their creation.

Actor Model

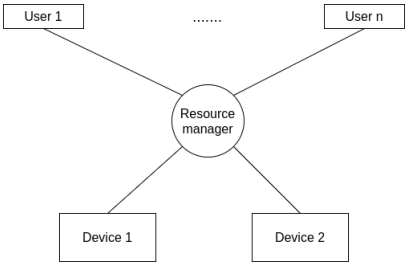


Figure: A static graph linking the resource manager to two devices and two users. Such graphs are an unsatisfactory representation of systems that may dynamically evolve to include new users or devices.

- Patterns of communication which are **possible** in an actor system define a **topology** on those actors.
- Each actor may, at any given point in its local time, **communicate** with some set of actors.
- As the computation proceeds, an actor may either communicate only with those actor that it could communicate with at the **beginning** of the computation, or it may **evolve** to communicate with other processes that it could not communicate with before.
- In the former case, the interconnection topology is said to be **static**, in the latter it is **dynamic**.

Defining an Actor System

Recursive factorial in Erlang

```
awaitFactorialProcess(N) ->
  Reply = self(),
  Pid = spawn(fun() -> factorial(Reply) end),
  Pid ! {compute, N},
  receive
    {result, X} ->
      X
  end.

factorial(Reply) ->
  receive
    {compute, N} when N == 0 ->
      Reply ! {result, 1};
    {compute, N} ->
      Cont = spawn(fun() -> factorialCont(N, Reply) end),
      self() ! {compute, N -1},
      factorial(Cont)
  end.

factorialCont(N, Reply) ->
  receive
    {result, X} ->
      Reply ! {result, N * X}
  end.
```

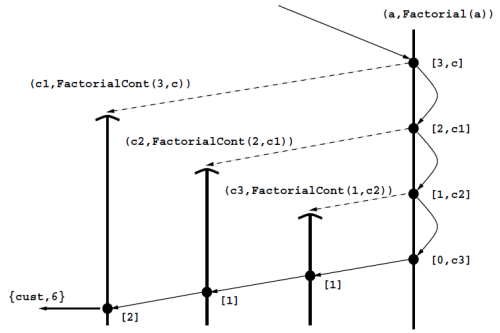


Figure: A recursive factorial computation. Each actor is denoted by a mail address and a behaviour. The FactorialCont represent the behaviour of the dynamically created customers (continuations).