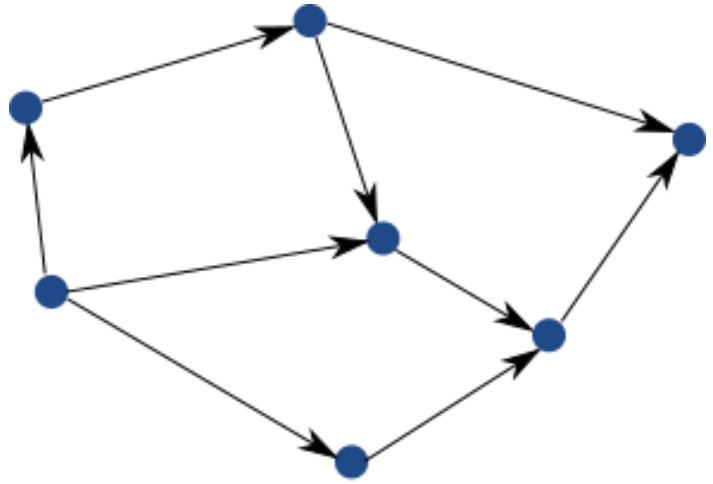
Lecture 13

Actor Model

The Actor Model

- A model of concurrent computation
- Main idea: *Everything is an Actor*
  - Similar to OO idea that *Everything is an Object*
  - An important difference between passing messages and calling methods is that messages have no return value.
- An actor can:
  - Send messages to other actors
  - React to messages it receives
  - Create new actors



Actors interacting with each other
by sending messages to each other

- There is no constraint on order between these
- Parallel computation and communication can occur in parallel across actors, also for any actor

History

- In computer science, the Actor model, first published in 1973 [Hewitt|Hewitt et al. 1973]-, is a mathematical model of concurrent computation. Many fundamental issues were discussed and debated in the early history of the Actor model.
- **Event orderings versus global state**
- A fundamental challenge in defining the Actor model is that it did not provide for global states so that a computational step could not be defined as going from one global state to the next global state as had been done in all previous models of computation.

From the definition of an Actor, it can be seen that numerous events take place: local decisions, creating Actors, sending messages, receiving messages, and designating how to respond to the next message received. Partial orderings on such events have been axiomatized in the Actor model and their relationship to physics explored.

The Actor Model

The Actor Model is a mathematical theory of computation that treats "*Actors*" as the universal conceptual primitives of concurrent digital computation

- was used:
  - as a framework for a theoretical understanding of concurrency, and
  - as the theoretical basis for several practical implementations of concurrent systems
- was been inspired
  - by physical laws
  - by languages as lambda calculus, Lisp, Simula, SmallTalk
  - Petri nets
- introduced patterns of passing messages for
  - Asynchrounous communication and control
- brought an important advance:
  - decoupling the sender from the communication

Characteristics

- inherent concurrency of computation within and among Actors
- dynamic creation of Actors
- inclusion of Actor addresses in messages
- interaction only through direct asynchronous message passing with no restriction on message reception order
- Enforce encapsulation without resorting to locks
- Use the model of cooperative entities reacting to signals, changing state, and sending signals to each other to drive the whole application forward

Characteristics

- Encapsulation is preserved by decoupling execution from signaling
  - method calls transfer execution, message passing does not.
- There is no need for locks.
  - Modifying the internal state of an actor is only possible via messages, which are processed one at a time eliminating races when trying to keep invariants.
- There are no locks used anywhere, and senders are not blocked.
  - Millions of actors can be efficiently scheduled on a dozen of threads reaching the full potential of modern CPUs.
  - Task delegation is the natural mode of operation for actors.
- State of actors is local and not shared, changes and data is propagated via messages, which maps to how modern memory hierarchy actually works.
  - In many cases, this means transferring over only the cache lines that contain the data in the message while keeping local state and data cached at the original core. The same model maps exactly to remote communication where the state is kept in the RAM of machines and changes/data is propagated over the network as packets.

Messages

An Actor can only communicate with another Actor to which it has an address

- **Message passing using types is the foundation of system communication:**

 Messages are the unit of communication

 A message can be sent to an address, which has a Type

- An **address** can have a meta-address to obtain additional information about the address, e.g., its type.
- Addresses can be implemented in a variety of ways:
  - direct physical attachment
  - memory or disk addresses
  - network addresses
  - email addresses

Specific differences

Actor Model assumes the following:

 the actor model was developed as an inherently concurrent model; the sequentiality is a special case that derived from concurrent computation

 Concurrent execution in processing a message.

 The following are *not* required by an Actor:

- a thread, a mailbox, a message queue, its own operating system process, *etc.*

For example, if an Actor is required to have a mailbox then, the mailbox would be an Actor that is required to have its own mailbox…

 Message passing has the same overhead as looping and procedure calling.

 Primitive Actors can be implemented in hardware.

Direct communication and asynchrony

- The Actor Model is based on one-way asynchronous communication. Once a message has been sent, the message is the responsibility of the receiver.
- Messages in the Actor Model are decoupled from the sender and they are delivered by the system on a best efforts basis.

Remark: Different from the previous approaches in which message sending is tightly coupled with the sender and sending a message synchronously transfers it on someplace, *e.g.*, to a buffer, queue, mailbox, channel, broker, server, *etc.* or to the "*ether*" or "*environment*" where it temporarily resides.

- The lack of synchronicity caused a great deal of misunderstanding at the time of the development of the Actor Model and is still a controversial issue.
- Because message passing is taken as fundamental in the Actor Model, there cannot be any requirement to use buffers, pipes, queues, classes, channels, *etc*.

=> a higher level of abstraction

Hardware or software implementation

- Implementations of the Actor Model typically make use of these hardware capabilities.
- there is no reason that the model could not be implemented directly in hardware without exposing any hardware threads, locks, queues, cores, channels, tasks, *etc*.
- Also, there is no necessary relationship between the number of Actors and the number threads, cores, locks, tasks, queues, *etc*. that might be in use.
- Implementations of the Actor Model are free to make use of threads, locks, tasks, queues, coherent memory, transactional memory, cores, *etc*. in any way that is compatible with the laws for Actors [Baker and Hewitt 1977].

Indeterminacy and quasi-commutativity

- indeterminacy <= the reception order of messages cannot be determined.
- quasi-commutativity – is the behavior affected by order in which operations (messages) occur?

Events and their orderings

- Activation ordering
  - one event activating another (there must be energy flow in the message passing from an event to an event which it activates).
- Arrival orderings
  - Arrival ordering is determined by *arbitration* in processing messages (often making use of a digital circuit called an arbiter).
  - The arrival events of an Actor are on its world line.
  - The arrival ordering means that the Actor model inherently has indeterminacy.
- Combined ordering
- Independence of the Law of Finite Chains Between Events in the Combined Ordering

Locality and Security

In processing a message:

an Actor can send messages only to addresses for which it has information:

1. that it receives in the message
2. that it already had before it received the message
3. that it creates while processing the message.

How to achieve performance

- Message passing has essentially same overhead as procedure calling and looping.
- Execution dynamically adjusted for system load and capacity (*e.g*. cores)
- Locality because execution is not bound by a sequential global memory model
- Inherent concurrency because execution is not bound by communicating sequential processes
- Minimize latency along critical paths

Robustness in Runtime Failures

- Runtime failures are always a possibility in Actor systems and are dealt with by runtime infrastructures.
- Message acknowledgement, reception, and response(a returned value or an exception) cannot be guaranteed although best efforts are made.
- Consequences are cleaned up on a best-effort basis.

Robustness is based on the following principle:

- If an Actor is sent a request, then the continuation *must* be one of the following two mutually exclusive possibilities:
    1. to process the response resulting from the recipient receiving the request
    2. to throw a Messaging exception

Specific implementations

- Actor systems implementations may use
  - Queues
  - Schedulers
  - Actors states

Actors are defined to have:

- A mailbox (the queue where messages end up)
- A behavior (the state of the actor, internal variables etc.)
- Messages (pieces of data representing a signal, similar to method calls and their parameters)
- An execution environment (the machinery that takes actors that have messages to react to and invokes their message handling code).
- An address.

Actor behavior

When an actor receives a message:

1.The actor adds the message to the end of a queue.

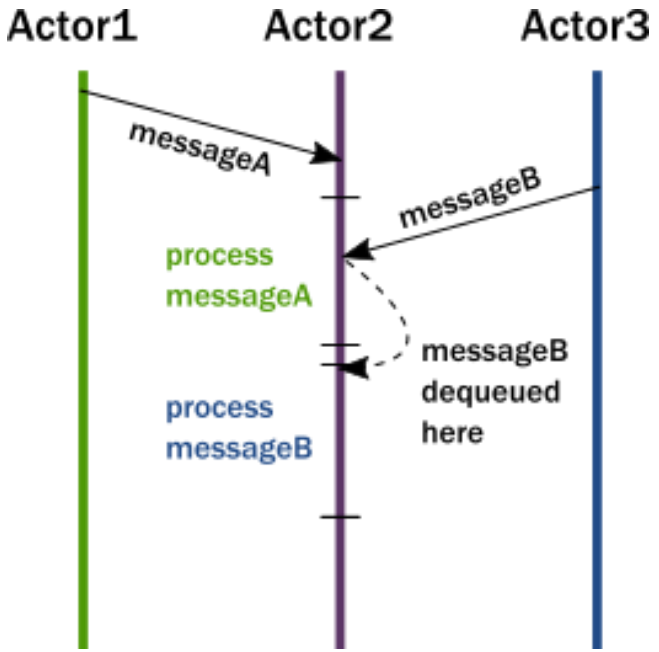
2.If the actor was not scheduled for execution, it is marked as ready to execute.

3.A (hidden) scheduler entity takes the actor and starts executing it.

4.Actor picks the message from the front of the queue.

5.Actor modifies internal state, sends messages to other actors.

6.The actor is unscheduled.



Some concrete implementations

- D
- Dart
- Elixir
- Erlang
- Ruby
- Scala
- Swift
- Akka (Java, Scala)
- PARLEY (Python)
- C++ Actor Framework(C++)