Actors

Paul Preißner

Fakultät für Informatik
Technische Universität München

July 6^{th} 2017

Brief history

- C. Hewitt et al. '73 onward: first theory of actor model, operational semantics, axioms
- W. Clinger '81: proved unbounded nondeterminism property
- G. Agha '85: formalization of semantic model
- Theoretical/Practical research by MIT, CalTech, industry, etc.
- Recent resurgence (strong relevance to distributed/cloud computing)

“A Model of Concurrent Computation in Distributed Systems“

- actors encapsulate computation (technically at any level)
- an actor may only send messages to actors it knows by name
- an (idling) actor receiving a message will accept it and execute the computation defined within, resulting in the possible actions:
  - sending new messages
  - creating new actors
  - updating its local state
- an actor can only influence its own local state

$\rightarrow$”self-contained, autonomous, interactive, asynchronously operating components” [Karmani, Agha]

Example structure

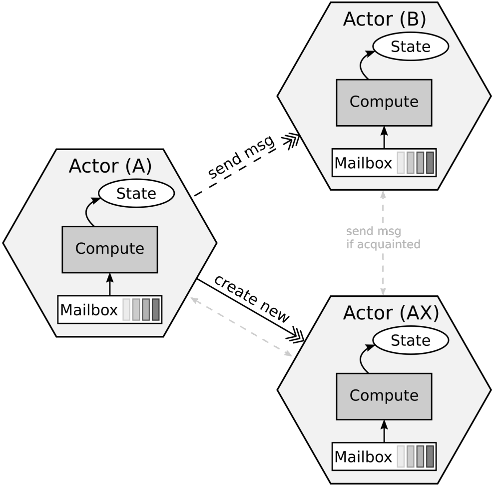


Figure: Any actor may send messages to other known actors, create new actors or update its own state. [inspired by Karmani, Agha]

(Example) Hello ...

using C++ Actor Framework; to illustrate:

```
[includes, usings]

behavior pong(event_based_actor* self, string selfname) {
  return {
    //if the message contains a string, proceed
    [=](const string& what) -> string {
      aout(self) << selfname << ": " << what << endl;
      // reply Pong
      return string("Pong!");
    }
  };
}
```

Specify behavior

(Example) Hello ...

```
void ping(event_based_actor* self, const actor& buddy, string selfname) {
  // send Ping to buddy (timeout for reply = 10s)
  self->request(buddy, std::chrono::seconds(10), "Ping!").then(
    //if the message contains a string, proceed
    [=](const string& what) {
      aout(self) << selfname << ": " << what << endl;
          //if reply is as expected, restart ping again
          if(what.compare("Pong...") == 0)
                  ping(self, buddy, selfname);
    }
  );
}
```

Specify actions

(Example) ... World!

```
int main() {
  [caf setup]
  // create a new actor that calls 'pong()'
  auto actor_B = system.spawn(pong, "B");
  // create another actor that calls 'ping(actor_B)';
  auto actor_A = system.spawn(ping, actor_B, "A"); }
```

Spawn actors and start something

```
B: Ping...
A: Pong...
B: Ping...
A: Pong...
B: Ping...
A: Pong...
[...]
```

Output (CAF "automatically" schedules to achieve high utilization across all 4 threads of i7 7500U)

Main semantic properties

- Encapsulation & atomic execution:
  actors don't share state;
  process one message at a time, arrivals mid-computation need to be buffered;
- Fairness
  every actor makes progress, every message is delivered eventually;
  assumes fair scheduler;
  →actor cannot stall entire program
- Location transparency
  physical location not bound to identifier;
  →(hidden) migration possible, i.e. *mobility* →allows load-balancing, efficiency optimization

Synchronization ...

Synchronization has to be through messages

- intuitively: "Remote Procedure Call"-*like* messaging
  send request, wait (and defer) until matching reply $\rightarrow$allows e.g. ordering
- local synchronization constraints: specify constraints on actors by which they accept or deny messages
- synchronizers: similar to pre-processing of messages

$\rightarrow$use methods to construct complex work patterns

… and abstraction

- use patterns to structure actors into different compositions
- use abstract compositions to group actors and allow more efficient identification or message recipients

Worst practices

- Faithful implementation of the model tends to be inefficient, shortcuts tend to be taken (that violate its properties)
  $\rightarrow$correctness of execution after optimization has to be checked

- Message latency: more distance = more latency
  $\rightarrow$use communication-computation overlap and suitable decomposition/migration to mask
- Naive send vs channels: individual sends may have high overhead
  $\rightarrow$utilize channels with stateful channel contracts to reduce overhead

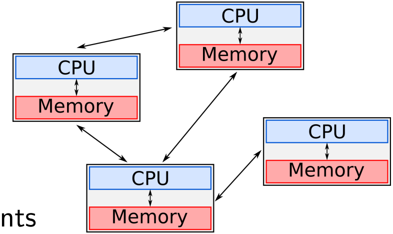
Worst practices

- Thread/switch overhead: the closer to a 1:1 map-to-thread mapping, the higher the overhead for switching execution to other actors; additional overhead for thread creation
  $\rightarrow$Use continuations based actors that don't perform full context switches and have reduced creation overhead
- Copying vs referencing: model demands no state be shared, naively this means no references be sent, only deep copies
  $\rightarrow$carefully allow send-by-reference for immutable types
- Scheduling: scheduler needs to guarantee fairness as in the model, not all schedulers satisfy this
  $\rightarrow$modify on a case-by-case basis; lazy thread creation - when tweaked - can be a relatively simple fix

Support

- $\sim$25 native actor languages
- $>$50 libraries for common languages (C, C++, C#, Java, JS, Python, Ruby, Haskell, LabVIEW, .NET, etc)
- most still actively supported, however some lacking proper documentation and developer support
- plug-ins for some IDEs exist (e.g. Erlang, Scala for Eclipse), some testing tools as well
- widespread support and documentation still spotty

in Distributed systems



- Naive: map actors directly to clients
- More suitable: map actors to program parts, e.g. individual services, users, etc →e.g. *Facebook* chat, *Twitter* queues, *Halo 4* services, *Lift* web-app framework
- HPC: actor model highly scalable (with care), e.g. ActorX10 project at TUM
- Microservices: composition of independent services

in Embedded systems

- in system level design: model components of the system as actors →model-based design (capture requirements of ES) →enables better abstraction, definition of interface, reasoning over execution
- on application level: e.g. ActorX10 project, model stages of algorithm as actors, pipeline execution to achieve high utilization →optimization of stages possible

Versus

- Petri nets: places (conditions), transitions (events), arcs between places$\leftrightarrow$transitions $\rightarrow$net of pre-/postconditions of possible events
  $\rightarrow$any number of tokens in the net, tokens trigger transitions and are transferred to postconditions
  $\rightarrow$actors modeled as units of computation, petri nets as transitions/events mostly independent of units

Versus

- process calculi: collection of formal models, processes interact by message-passing (anonymous, through channels), processes as composition of primitives and operators subject to algebraic laws
  $\rightarrow$message-passing similar to actors, processes could be modeled to resemble actors
- I/O automata: automaton modeled into any kind of single component, as state machine with in-/output and (hidden) internal actions
  $\rightarrow$similar to encapsulation in actors, automata can be modeled to resemble actor model structure

Conclusion?

- actor model has great potential in achieving scalable concurrency
- quantitative and qualitative developer support is lacking, but increasing
- more and more use in large and small, distributed and embedded, commercial and free/open projects

Q&A