Programmation Concurrente

CM7 - The Actor Model

Florian Brandner & Laurent Pautet
LTCI, Télécom ParisTech, Université Paris-Saclay

Outline

Course Outline

- CM1: Introduction
- CM2: The *shared-memory* model
- CM3-6: Concurrent programming POSIX/Java (L. Pautet)
  Patterns and Algorithms (L. Pautet)
- **CM7: Actor-based programming (me)**
  - Definition of actors
  - Actor primitives and concepts
  - Brief introduction to Akka Actor Library
- CM8: Transactional memory (me)

Recapture

Shared-Memory Model

- All processors/threads share the same main memory
  - Data is exchanged through that memory
  - Data is shared through that memory
  - Threads synchronize through that memory
- Concurrent accesses
  - Might cause some troubles
  - Coherency: how do threads agree on the *latest* value?
  - Consistency: in which order do updates appear *globally*

  $\Rightarrow$ Memory models cover both aspects

Shared-Memory Model

- All processors/threads share the same main memory
  - Data is exchanged through that memory
  - Data is shared through that memory
  - Threads synchronize through that memory
- Concurrent accesses
  - Might cause some troubles
  - Coherency: how do threads agree on the *latest* value?
  - Consistency: in which order do updates appear *globally*

  $\Rightarrow$ Memory models cover both aspects

Is this the only model?

Sockets

Do not require shared memory:

- Allow to send messages over a network
  - Various protocols possible (UDP, TCP/IP, ...)
  - Receiver has to *listen* for messages (`recv`, `recvfrom`)
  - Similar interface as regular files
- Name vs. addresses
  - Machine names to find receiver (`gethostbyname`)
  - No common naming of services (port numbers)
- Available almost everywhere (C, Java, ...)
  - But cumbersome to use
  - Can we do better?

Sockets

Do not require shared memory:

- Allow to send messages over a network
  - Various protocols possible (UDP, TCP/IP, ...)
  - Receiver has to *listen* for messages (`recv`, `recvfrom`)
  - Similar interface as regular files
- Name vs. addresses
  - Machine names to find receiver (`gethostbyname`)
  - No common naming of services (port numbers)
- Available almost everywhere (C, Java, ...)
  - But cumbersome to use
  - Can we do better?

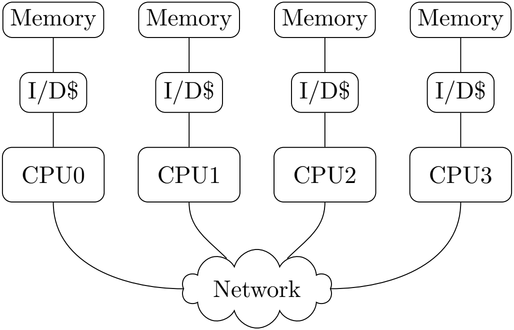
Can we generalize this?

The Message-Passing Model

Message Passing between Computers

General concept, with many different implementations:

- Several processors in potentially many computers
- No globally shared memory accessible to all processors
- Information exchange is based on *messages*



Main Issues

Very different problems (w.r.t. shared-memory model):

- How is data exchanged?
  (point-to-point messages, broad-, multi-cast)
- How are data and computations distributed?
  (computations impossible without data)
- How can one balance the load between computations at processors and the communication over the network?

$\Rightarrow$ Coarser form of parallelism due to cost of communication
$\Rightarrow$ Almost exclusively controlled by programmer
(few tools available)

Implementations

Several programming frameworks/languages are based on message passing:

- **OpenMPI** (http://www.open-mpi.org/)
  C, C++, Fortran library for large-scale parallel computing using message passing (often used in scientific computing)
- **Erlang** (http://www.erlang.org/)
  Old functional programming language that was recently rediscovered. Parallelism is based on actors (*computations*), which exchange information through message passing.
- Stream programming (StreamIt) and synchronous programming languages (Lustre, Esterel, SCADE)
- …

The Actor Model

Actors

Basic unit of computation:

- Actors can **communicate** among each other
- Actors can **compute** in response to a message
- Actors can **create** other actors
- Actors can designate how to handle the next message

Definition goes back to Carl Hewitt (70's), and was later refined by Gul Agha (80's).

Actors (2)

Additional features:

- Each actor has a unique name
- Each actor has its own private state (no global state)
- Pending messages are kept in a mailbox[1] (treated later)

[1] This is optional

Communication

Weak guarantees concerning communication

- Communication is one-way and asynchronous
  (neither sender nor receiver is blocked)
- Messages are delivered in best-effort manner
  (messages may be lost or delayed infinitely)
- Message order is not defined, except:
  - A message is sent before it can be received
  - Even for messages of the same actor
  - No other guarantees
- Actors may communicate names of actors

Computation

Messages are handled *atomically* by actors

- Raise level of abstraction when reasoning about actors
- Instead of *micro steps* (instructions) ...
- ... use *macro steps* (handling of messages)
- Absence of global state simplifies things

TELECOM ParisTech

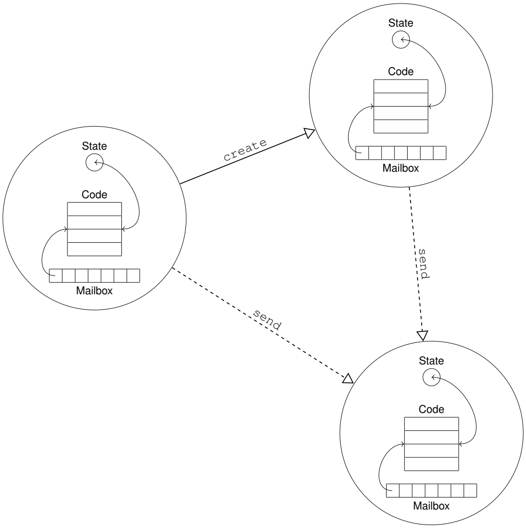
How is an Actor implemented?

Simplistic model based on pseudo code:

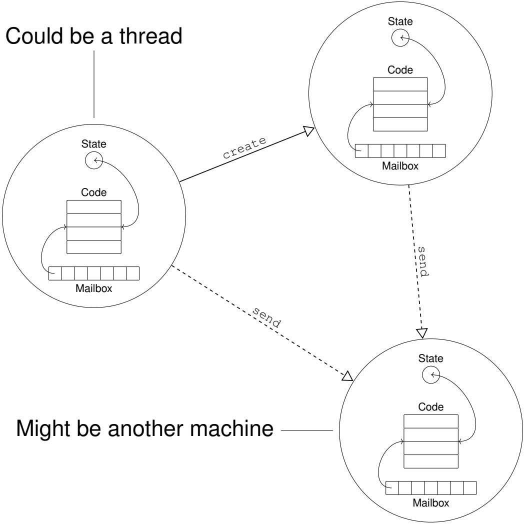
```
while (true)
{
  while (!mailbox.empty())
  {
    // actor can chose which message to handle
    // next (e.g., through priorities)
    msg = mailbox.select_next_message()

    // perform action depending on message type
    switch (msg)
    {
      // create actors, send messages, compute, ...
      case ...
      case ...
      case ...
    }
  }
}
```

Example: Actors



Example: Actors



Actor Semantics

Encapsulation and Atomicity

- Actors do not share state
  - Data is exchanged using messages
  - Data is effectively copied
- Actors handle one message at a time
  - Many actors may work concurrently
  - However, each actor only processes one message at a time
  - Message processing appears atomic for external observers

This ensures the absence of race conditions on variables (deadlocks due to message processing are still possible).

Fairness

- An actor makes progress whenever it has some computation to do
- An actor processes one of pending messages otherwise
  - Actors may still select which message to process next
  - This can be used to prioritize message processing

This ensures global progress of the entire system.

Location Transparency

- The location of an actor does not affect its name
- The location of an actor does not affect message passing
- After termination of an actor another may take its name
  - This is useful, for instance, to restart crashed actors
  - This can also be used to migrate actors from one location to another

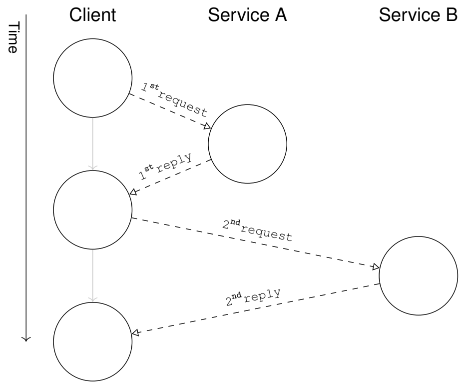
This improves robustness and portability.

Communication Patterns

RPC-like Requests

Messages are one-way, thus

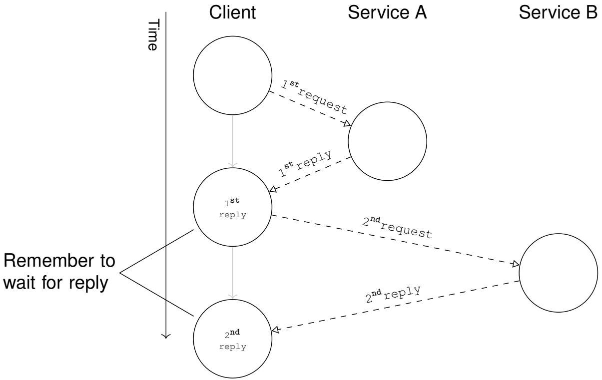
- Client and server need to send messages back and forth
- Client has to remember that it waits for a reply
- This is similar to remote procedure calls (RPC)



RPC-like Requests

Messages are one-way, thus

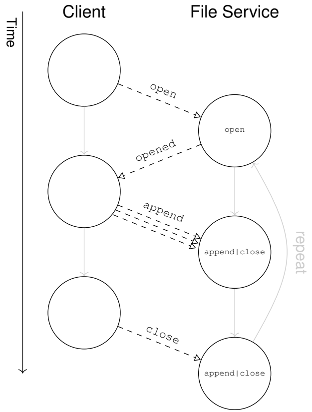
- Client and server need to send messages back and forth
- Client has to remember that it waits for a reply
- This is similar to remote procedure calls (RPC)



Local Message Constraints

Messages acceptance may depend on history:

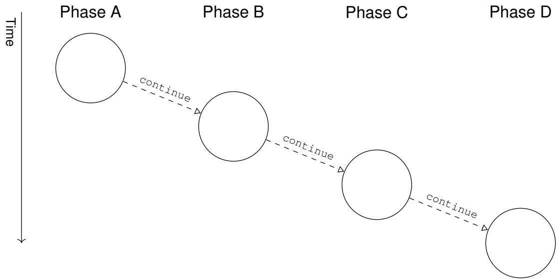
- Actor may expect specific *sequences* of messages
- Message acceptance may thus depend on the actor's state
- Predicates and message filters can be applied to mailbox



Pipelining

Handle message sequences in parallel:

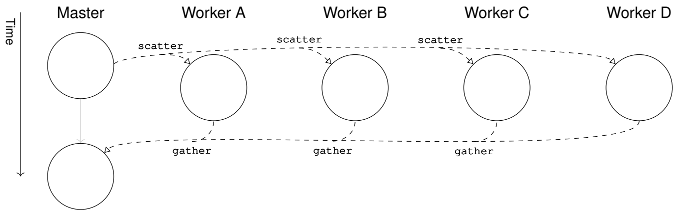
- Similar to the idea of piplined processors (SE201)
- Cascade of actors, each handling a step of the sequence
- All of these actors work in parallel



Divide and Conquer/Map-Reduce

Popular parallelization technique:

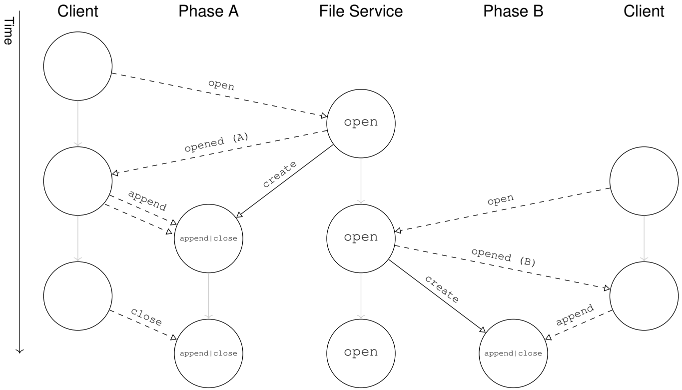
- Divide work into smaller pieces
- Scatter pieces to worker actors for processing
- Gather replies to constitute final answer



Combining Patterns

Patterns can of course be combined:

- RPC-like requests combined with local constraints and pipelining
- Note: the RPC request creates the second pipeline step



The Akka Actor Library

Akka Actor Library

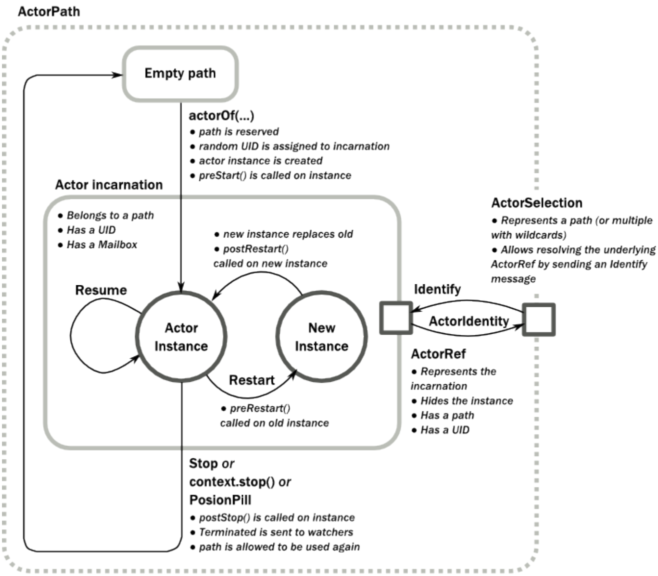
- Allows to implement actors in Java (also Scala)
- Provides naming service (to find actors)
- Provides communication service (for message passing)
- Provides utility functions and classes
  (fault-tolerance, watchdogs, ...)

- Akka is open-source:[2]

  http://akka.io/

[2] We use version 2.3.14 for Java 6.

Actor Lifecycle in Akka



Defining Actors

Simply by defining a new class:

```
import akka.actor.UntypedActor;

public class MyActor extends UntypedActor {
  @Override
  public void onReceive(Object msg) {
    // code goes here. for now, ignore all messages
    unhandled(msg);
  }
}
```

http://doc.akka.io/docs/akka/2.3.14/java/untyped-actors.html

Instantiating an Actor

First, define a `Props`, i.e., a kind of receipt:

```
import akka.actor.UntypedActor;
import akka.actor.Props;
import akka.japi.Creator;

public class MyActor extends UntypedActor {
  public static Props props() {
    return Props.create(new Creator<MyActor>() {
      private static final long serialVersionUID = 1L;

      @Override
      public MyActor create() throws Exception {
        return new MyActor();
      }
    });
  }
}
```

http://doc.akka.io/japi/akka/2.3.14/akka/actor/Props.html

Instantiating and Initializing an Actor

Then, instantiate the actor in the current *context*:

```
import akka.actor.UntypedActor;
import akka.actor.Props;
import akka.japi.Creator;
import akka.actor.ActorRef;

public class MyActor extends UntypedActor {
  @Override
  public void preStart() {
    // actually create actor and obtain an actor reference
    final ActorRef actorInstance =
                    getContext().actorOf(MyActor.props());

    // send a message to the newly instantiated actor
    actorInstance.tell(1, getSelf());
  }
}
```

http://doc.akka.io/japi/akka/2.3.14/akka/actor/Actor.html

Start-up of Akka Applications

Instantiate a first actor, which then takes over:

```
import akka.actor.UntypedActor;

public class MyActor extends UntypedActor {
  public static void main(String[] args) {
    // Simply tell Akka to create an actor, which takes over
    akka.Main.main(new String[] { MyActor.class.getName() });
  }
}
```

http://doc.akka.io/japi/akka/2.3.14/akka/Main.html

Actor Selection

Find actors using their names/paths:

```
import akka.actor.UntypedActor;
import akka.actor.ActorSelection;

public class MyActor extends UntypedActor {
  public void mySendTo(String pattern) {
    // find all actors matching the pattern
    ActorSelection selection =
                    getContext().actorSelection(pattern);

    // send the same message to all of the selected actors
    selection.tell(2, getSelf());
  }
}
```

http://doc.akka.io/japi/akka/2.3.14/akka/actor/ActorSelection.html

Message Passing

- Messages should be *immutable*
  (Java does not allow to enforce this, so its merely a convention)
- Three distinct primitives for sending:
  - Non-blocking without reply (`tell()`)
  - Non-blocking providing a reply through a *future* (`ask()`)
  - Forwarding of messages (`forward()`)
- Message retrieval:
  - Automatically handled by Akka (`onReceive()`)
  - Signal unexpected messages (`unhandled()`)

TELECOM ParisTech

Additional Functions

Some utility functions

- Use actor reference (`ActorRef`) to manipulate (other) actors:
  - Get reference to current actor (`getSelf()`)
  - Get sending actor of current message (`getSender()`)
  - Get actor path and name (`path().name()`)
- Use context to interact with the environment:
  - Create actors (`getContext().actorOf()`)
  - Terminate actors (`getContext().stop()`)
  - Get parent actor (`getContext().parent()`)
  - Get child actors (`getContext().children()`)
- More functions:
  - Fault tolerance (monitoring, hot-swapping, watchdogs, . . .)
  - Message passing (routing, dispatching, mailboxes, . . .)
  - . . .

Summary

- Brief introduction to the Actors Model:
  - Basic unit of computation
  - Only has private state
  - Reacts to incoming messages
  - Can create actors, compute, and send messages
  - Only communicates via messages (no global/shared state)
- Principles:
  - Encapsulation and atomicity
  - Fairness
  - Location transparency
- Communication patterns:
  - RPC-like requests (in Akka: `ask()`)
  - Local message constraints (in Akka: `stash()`)
  - Pipelining (in Akka: `pipe()`)
  - Divide and conquer (map/reduce)
- Introduction to the Akka Actor Library

Further Reading

- Actor Model of Computation: Scalable Robust Information Systems Carl Hewitt (arxiv, 2010-2015)
- Actors
  Rajesh K. Karmani and Gul Agha (Encyclopedia of Parallel Computing, 2011)
- Actors: A Model for Reasoning about Open distributed Systems Gul Agha, Prasannaa Thati, Reza Ziaei (Formal Methods for Distributed Processing: A Survey of Object-Oriented Approaches, 2001)