eConcurrency
Concurrent and Distributed Programming in OCaml

Benedikt Grundmann

August 16th, 2007

Problem overview

- OCaml gives us concurrent, but not parallel programming!

Problem overview

- OCaml gives us concurrent, but not parallel programming!
- Shared memory multi threading

Problem overview

- OCaml gives us concurrent, but not parallel programming!
- Shared memory multi threading
- No distribution

Basic idea

- Each concurrent entity is one operating system process

Basic idea

- Each concurrent entity is one operating system process
- Communicate using message passing

Message passing

Two primitives:

```
val send : message -> unit

val receive : unit -> message
```

To be in sync or not to be

To be in sync or not to be

To be in sync or not to be

Mailboxes, Addresses...

```
type mailbox
type address
```

Mailboxes, Addresses...

```
type mailbox
type address

val mailbox : unit -> mailbox
val send : address -> message -> unit
val receive : mailbox -> message
```

What is a message?

What is a message?

Application specific!

What is a message?

Application specific!

```
type 'a mailbox
type 'a address
val send : 'a address -> 'a -> unit
val receive : 'a mailbox -> 'a
```

Message

Implementation: serialize/deserialize messages

Message

Implementation: serialize/deserialize messages

```
type 'a msg
val serialize : 'a msg -> 'a -> string
val deserialize : 'a msg -> string -> 'a
```

Message

Implementation: serialize/deserialize messages

```
type 'a msg
val serialize : 'a msg -> 'a -> string
val deserialize : 'a msg -> string -> 'a

val mailbox : 'a msg -> 'a mailbox
```

Let's use it

```
type message server_reply   = string
type message client_request = string
```

Let's use it

```
type message server_reply   = string
type message client_request = string

let client name =
  let mb = mailbox server_reply_msg in
  send server_address "Bene";
  let r = receive mb in
  print_endline r
```

Let's use it

```
type message server_reply   = string
type message client_request = string

let client name =
  let mb = mailbox server_reply_msg in
  send server_address "Bene";
  let r = receive mb in
  print_endline r

let rec server () =
  let mb = mailbox client_request_msg () in
  server' mb

and server' mb =
  let name = receive mb in
  send client_address (sprintf "Hello %s" name);
  server' mb
```

Sending addresses

```
type message server_reply   = string
type message client_request =
  server_reply address * string
```

Sending addresses

```
type message server_reply   = string
type message client_request =
  server_reply address * string

let server' mb =
  let client_address, name = mb in
  send client_address (sprintf "Hello %s" name);
  server' mb
```

Sending addresses

```
type message server_reply   = string
type message client_request =
  server_reply address * string

let server' mb =
  let client_address, name = mb in
  send client_address (sprintf "Hello %s" name);
  server' mb

let client name =
  let mb = mailbox server_reply_msg in
  send server_address (address_of mb, "Bene");
  let r = receive mb in
  print_endline r
```

Server, where are you?

```
type globalpid
```

Server, where are you?

```
  type globalpid

# to_string gpid
- : string = "ec://134.96.60.197/6656-272303C7"
```

Server, where are you?

```
  type globalpid

# to_string gpid
- : string = "ec://134.96.60.197/6656-272303C7"

  val to_string : globalpid -> string
  val of_string : string -> globalpid
```

Server, Do you understand me?

```
val mailbox : ?name:string -> 'a msg -> 'a mailbox
```

Server, Do you understand me?

```
val mailbox : ?name:string -> 'a msg -> 'a mailbox

val resolve : msg:'a msg -> pid:globalpid
      -> string -> 'a address (* Not_found *)
```

Server, Do you understand me?

```
val mailbox : ?name:string -> 'a msg -> 'a mailbox

val resolve : msg:'a msg -> pid:globalpid
      -> string -> 'a address (* Not_found *)

val signature : 'a msg -> string
```

Echoserver, continued

```
let rec server () =
  let mb = mailbox ~name:"echoserver"
             client_request_msg in
  server' mb
```

Echoserver, continued

```
let rec server () =
  let mb = mailbox ~name:"echoserver"
             client_request_msg in
  server' mb

let client server_gpid name =
  let mb = mailbox server_reply in
  try
    let server_address = resolve
        ~msg:client_request ~pid:server_gpid
        "echoserver" in
    send server_address (address_of mb, "Bene");
    let r = receive mb in
    print_endline r
  with Not_found ->
    prerr_endline "ERROR!";
    exit 1
```

Errors?

- Asynchroneous error handling is hard!

Errors?

- Asynchroneous error handling is hard!
- Conclusion: We do not handle errors

Errors?

- Asynchroneous error handling is hard!
- Conclusion: We do not handle errors
- Handshake

Errors?

- Asynchroneous error handling is hard!
- Conclusion: We do not handle errors
- Handshake
- Timeout

```
val receive : ?timeout:float
    -> 'a mailbox -> 'a
```

Errors?

- Asynchroneous error handling is hard!
- Conclusion: We do not handle errors
- Handshake
- Timeout

```
val receive : ?timeout:float
    -> 'a mailbox -> 'a
```

- Heartbeat protocol

Heartbeat

```
val observe : globalpid -> unit

val observed : (globalpid * reason) mailbox
```

Heartbeat

```
val observe : globalpid -> unit

val observed : (globalpid * reason) mailbox

type reason = [ `Exit | `Crash ]
```

Mailbox Combinators

```
val map : ('a -> 'b) -> 'a mailbox -> 'b mailbox
val multiplex : 'a mailbox list -> 'a mailbox
```

Mailbox Combinators

```
val map : ('a -> 'b) -> 'a mailbox -> 'b mailbox
val multiplex : 'a mailbox list -> 'a mailbox

let server () =
  let mb = mailbox ~name:"echoserver"
                   client_request_msg in
  let mb' = multiplex [
              map (fun r -> Request r) mb;
              map (fun e -> Exit e) observed
            ] in
  server' mb

and server' mb =
  match receive mb with
    | Exit e -> ...
    | Request (client_address, name) ->
        ...
```

Questions?