JOCAML

A JOIN-CALCULUS IMPLEMENTATION IN OCAML

Locations -- agents and sites

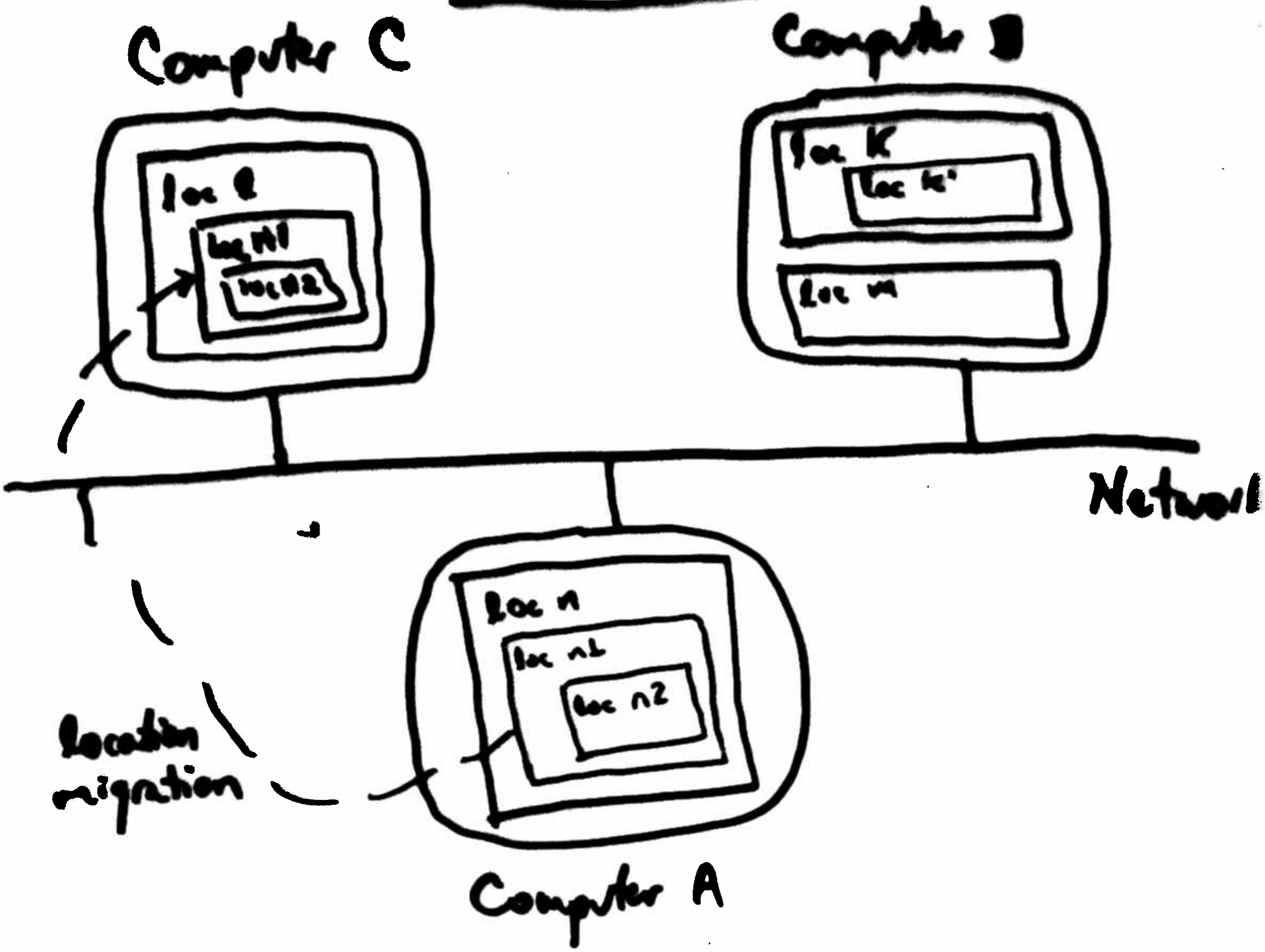
Channels -- communication links

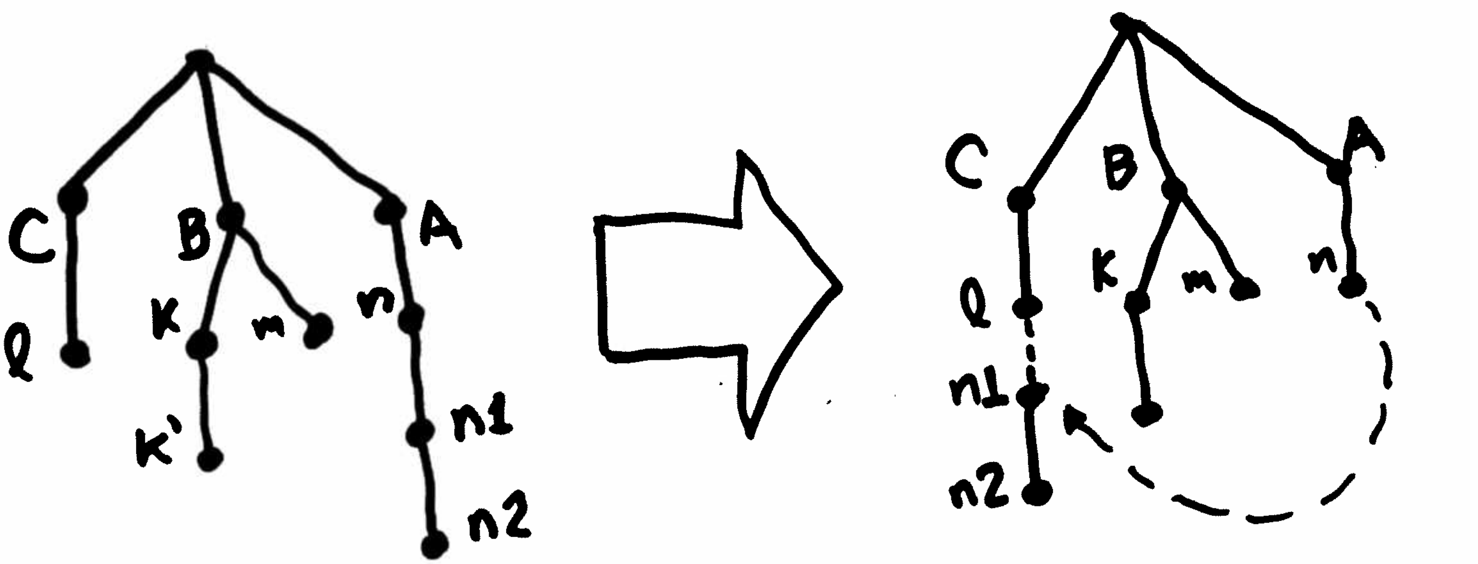
Locations organized in hierarchical tree:

- Sites are top-level locations or root

- Agents are nested locations.

MIGRATION


Computer C
loc l
loc n1
loc n2
Computer B
loc k
loc k'
loc m
Network
loc n
loc n1
loc n2
location migration
Computer A


C
B
A
l
k
m
n
k'
n1
n2
C
B
A
l
k
m
n
n1
n2

CHANNELS

Created by definitions: channels associate messages to processes, in an assymmetric way:

Messages sent on a channel are all received by the same channel definition.

Communication links are kept during a[illegible] migrations, i.e. messages sent on chan[illegible] are transparently forwarded to the channel definition, from any age[illegible]

CHANNEL TYPES

Asynchronous channels

Synchronous channels (a.k.a. RPC)

---

go primitive triggers migration.

fail guard only returns when location is halted or crashed.

Starting Communications

Ns.register — to register name values in the name server.

Ns.lookup — to request values from their names.

Typechecking

OCAML does static type-checking.

In open systems, dynamic types are used at application interfaces.

vartype — provides type info checked at name server.

EXAMPLE

```
(* SERVER *)

# let host = Unix.gethostname () ;;
  Ns.register "server host" host vartype ;;

val host : string = "here.cs.rpi.edu"
Warning: VARTYPE replaced by type
    string metatype
Using multicast
Querying name server here.cs.rpi.edu:20001
- : unit = ()
```

EXAMPLE (continued)

```
(* CLIENT *)

# let host : string = Ns.lookup "server host"
      vartype;;
print_string ("The server is on "^host);;

Warning: VARTYPE replaced by type
    string metatype
Using multicast
(Querying name server here.cs.rpi.edu:20001
val host : string = "here.cs.rpi.edu"
The server is on here.cs.rpi.edu-: unit = ()
```

PROCESSES

- OCAML uses expressions, i.e. terms computed to a value.
- JOCAML process execution yields no value: Used for side-effects, mainly communication & synchronization

{ . . . } syntax

↑ expressions and sub-processes.

P | Q for parallel process composition.

spawn { P } to create new thread executing process P.

CHANNELS

- Asymmetric links between one receiver and multiple senders.
- Channel names can be exchanged.
- Receiver agent can migrate.
  ⟹ Semantics remains the same.

SYNCHRONOUS CHANNEL DEFINITION

let def <u>name</u>(args) = P(args)

A new thread gets created per msg received

reply statement to return a value.

ASYNCHRONOUS CHANNEL DEFINITION

let def name<u>!</u> (args) = P(args)

PRINTER EXAMPLE

```
(* SERVER *)

let def print s = {
        print_string s; reply };;
Ns.register "Printer" print vartype;;
```

```
(* CLIENT *)

let printer = Ns.lookup "Printer" vartype;;
printer "Hello ";;
printer "I'm the client";;
```

JOIN PATTERNS

```
let def u(a) | v(b) = P(a,b)
     or u(a) | w(b) = Q(a,b)
```

If only one message is present on u, P and Q will never happen in parallel.

π-Calculus Channels

```
let def new_pi_channel () =
    let def send x | receive () =
        reply x to receive
      | reply to send
    in reply send, receive ;;
val new_pi_channel : unit -> ('a -> unit) * (unit -> 'a)
```

PRINTER EXAMPLE WITH RENDEZ-VOUS

```
(* SERVER *)
let def print! (str, cont) | lock! () =
   { print_string str ;
     { cont() | lock () } } ;;
spawn { lock () } ;;
Ns.register "Printer" print vartype ;;
```

```
(* CLIENT *)

let printer = Ns.lookup "Printer" vartype ;;
let def cont1!() | cont2!() | cont3!() =
      { exit 0; }
spawn { printer ("Hello", cont1) |
        printer ("beautiful ", cont2) |
        printer ("world", cont3) }
```

REFERENCE CELL IN JOCAML

```
let def create_ref v0 =
    let def state! v | get () =
                  state v | reply v
          or state! v | set new_v =
                  state new_v | reply
    in state v0 | reply get, set ;;
val create_ref : 'a -> (unit -> 'a) * ('a -> unit)
                                            = <fun>
```

Reference Cell Usage in Jocaml

```
# let g0,s0 = create_ref 0
  and g1,s1 = create_ref "" ;;

val g0 : unit -> int = <fun>
val s0 : int -> unit = <fun>
val g1 : unit -> string = <fun>
val s1 : string -> unit = <fun>

# print_int (g0 ());;
0-: unit = ()
# s0 5 ;;
-: unit = ()
# print_int (g0 ());;
5:- unit = ()
```

Agents

Sites, agents, and groups of agents in JOCAML share a single abstraction: the location.

let loc agent do process

creates a new agent, named agent, executing the thread process.

Agent Example

```
(* SERVER *)

let def print s = print_string s; reply;;
let target = Ns.lookup "client" vartype;;

let loc agent do {
    print ("I'm on computer "^
            Unix.gethostname());
    go target;
    print ("I'm on computer "^
            Unix.gethostname());
    print_string "Hello";
};;
```

AGENT EXAMPLE (continued)

```
(* CLIENT *)

let loc target do {} ;;

Ns.register "client" target vartype ;;
```

COMMUNICATING AGENTS

let loc agent
def channels
do { process }

Channels are unidirectional. These channel definitions allow agents to receive messages.

Channel bindings never change: communication is independent of the location of sending or receiving agents.

COMMUNICATING AGENT EXAMPLE (1)

```
(* SERVER *)

let def create_speaker (l, name) =
    let loc agent
        def print! (s, send) = {
            print_string s ; speak (send) }
        and speak! send = {
            print_string (name ^ ">");
            let s = read_line () in
            send (s, send) }
      do { go l; }
  in { reply print, speak } ;;
```

COMMUNICATING AGENT EXAMPLE (2)

```
(* SERVER CONTINUED *)

let def talk!(l1,n1) | no_client!() =
            one_client(l1,n1)
    or talk!(l2,n2) | one_client(l1,n1) =
      { let print1, speak1 =
              create_speaker (l1,n1) in
        let print2, speak2 =
              create_speaker (l2,n2) in
        speak1 (print2) | no_client ()
           speak2 (print1) ?
      };;
spawn {no_client()};;
Ns.register "talk" talk vartype;;
```

Communicating Agent Example (3)

```
(* CLIENT *)
let username = getenv "USER";;
let talk = Ns.lookup "talk" vartype;;
let loc client do {
        talk (client, username) };;
```

FAILURE MODEL IN JOCAML

- halt () primitive process, atomically halts every process inside of this location.

- fail there;P primitive detects that location "there" halted and runs P.

- Fail is detected (in current JOCAML implementation) only in same run-time. It is also triggered if run-time containing halting location becomes unreachable or terminates.

FAILING AGENT EXAMPLE

```
let loc agent
  def say s = print_string s; reply
  do { halt(); };;
spawn { say "it may work before.\n"; };;
spawn { fail agent;
        print_string "location stopped";
        say "it never works after\n"; };;

val agent: Join.location
val say : string -> unit
=> the location stopped
```

LOCATIONS AND BINDING EXAMPLE (1)

```
# let def cell there =
    let def log s =
        print_string (s); reply in
    let loc applet
        def get() | some!x
            = log("empty"); none()
            | reply x
        and put x | none!()
            = log("contains "^x);
              some x | reply
    do { go there; none() } in
  reply get, put ;;
```

LOCATIONS AND BINDING EXAMPLE (2)

```
Ns.register "cell" cell vartype;
Join.server()

# let cell = Ns.lookup "cell" vartype

let loc user
  do {
    let get, (put:string->unit) =
      cell user in
      put "world";
      put ("hello " ^ get());
      print_string (get());
      exit 0;
  }
```