CSCI.6500/4500 Distributed Computing over the Internet—Programming Distributed Computing Systems (Varela)—Sections 10.1.2, 10.2.2, 10.4.4

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2021
Distributed and Mobile Systems with JoCaml

• JoCaml follows the distribution and mobility model of the distributed join calculus (Fournet et al., 1996).

• This includes the notion of a tree of locations with creation, migration, termination, and failure detection capabilities.
Distributed Systems with JoCaml

JoCaml concurrent programs can be directly used for distributed systems programming.

- Since the language entirely relies on asynchronous communication (synchronous communication is modelled as asynchronous communication and an implicit continuation,) JoCaml programs can run on a single machine or on several machines transparently.

- Channel names can be used as part of both local and remote messages and locality is transparent.
Distributed Systems: Name Service and Sites

JoCaml’s distribution model consists of

- a name server (Join.Ns) and
- sites (Join.Site.)

Sites are bytecode executables generated by the compiler and linked to the distributed run-time.
Distributed Systems: Name Service

JoCaml values, including channel names, can be registered and looked up in the name server using plain strings. For example, a \textit{square} function can be created and registered as follows:

```jocaml
spawn begin
  def f (x) = reply x*x to f in
  Join.Ns.register Join.Ns.here "square" (f: int -> int);
  Join.Site.listen (Unix.ADDR_INET
    (Join.Site.get_local_addr(),12345));
  def x () & y () = reply to x in x;
  0
end ;;
```
The `Join.Site.listen` function creates a TCP/IP socket to wait for incoming connections. Here, we assume that the program is executed in the server `a.wcl.cs.rpi.edu`.

The line defining channels `x` and `y` is a way to make the program wait indefinitely without ever halting (since no other process could possibly write on `y` because of `y`'s lexical scope.)
Distributed Systems: Name Service

Another process (in a different run-time) that looks up and uses the \textit{square} function can be created as follows:

```ocaml
spawn begin
  let server =
    let server_addr = Unix.gethostbyname "a.wcl.cs.rpi.edu" in
    Join.Site.there (Unix.ADDR_INET (server_addr.Unix.h_addr_list.(0),12345))
  in
  let ns = Join.Ns.of_site server in
  let sqr = (Join.Ns.lookup ns "square" : int -> int) in
  print_int (sqr 2);
end;;
```
let server =
    let server_addr = Unix.gethostbyname "a.wcl.cs.rpi.edu" in
    Join.Site.there
        (Unix.ADDR_INET
            (server_addr.Unix.h_addr_list.(0),12345))
    in
    let ns = Join.Ns.of_site server in
    let sqr = (Join.Ns.lookup ns "square" : int -> int) in
    print_int (sqr 2);

• The program first connects to a.wcl.cs.rpi.edu:12345 with the function Join.Site.there to get the abstract value server which represents the run-time on a.wcl.cs.rpi.edu.

• The name server is then extracted with the library function Join.Ns.of_site and queried with Join.Ns.lookup.

• After this operation sqr is an alias for the remote channel f.
Mobile Systems

Regarding mobility, JoCaml introduces the notion of *locations* which encapsulate a set of definitions and running processes.

- Locations have a name that
  - can be communicated in messages,
  - registered in the name server, and
  - used as arguments to primitives that dynamically control how they are related.
Mobile Systems Example

A location containing a square definition can be written as follows:

```plaintext
let loc this_location
    def square (x) = reply x*x
do {
    print_int (square 2);
0
} ;;
```

This location can be registered in the name server as follows:

```plaintext
spawn begin
    Join.Ns.register Join.Ns.here "sq_location"
        (this_location: Join.location);
    Join.Site.listen (Unix.ADDR_INET
        (Join.Site.get_local_addr(),12345));
    def x () & y () = reply to x in x;
0
end;;
```
Mobile Systems

Distributed computations are organized as trees of nested locations.

- Processes and definitions are attached to a location and once the location is created, no new bindings and processes may be made in the location from outside the location.

- Mobility is modeled as the movement of a location from one enclosing location to another.

- Similarly to mobile ambients, location movement is subjective, i.e., triggered by a process inside the location to be moved.
Mobile Systems Example

A mobile agent that goes to the site where the square definition is located, can be written as follows:

```ocaml
let loc mobile
  do {
    let sq_loc =
      (Join.Ns.lookup ns "sq_location" : Join.location) in
    go sq_loc;
    let sqr = (Join.Ns.lookup ns "square" : int -> int) in
    print_int (sqr 2);
  }
```
Applet Server Example

An applet server can be encoded in JoCaml as follows:

```ocaml
def cell there =
  def log s = print_string ("cell " ^ s ^ "\n");
  flush stdout;
  reply to log

in
  let loc applet
def get () & some(x) = log ("is empty"); none() &
    reply x to get
  and put(x) & none() = log ("contains " ^ x); some(x) &
    reply to put
  do { go there; none (); 0 } in
  reply get, put ;;

spawn begin
  Join.Ns.register Join.Ns.here "cell"
    (cell: Join.location Join.chan);
  Join.Site.listen (Unix.ADDR_INET
    (Join.Site.get_local_addr(),12345));
  def x () & y () = reply to x in x;
  0
end;;
```
Applet Server Example

A cell applet has two states:

• either none() or some(x), and
• two methods get and put.

For each request, the cell applet server

• first creates a new cell applet,
• migrates it to the given there client location, and then
• returns a reference to the applet methods.

Notice that the applet can communicate from the client back to the server through the log channel within its lexical scope.
Applet Client Example

An applet client can be written as follows:

```ocaml
let cell =
    (Join.Ns.lookup ns "cell" : Join.location Join.chan) in

let loc user
    do {
        let get,put = cell user in
        put("world");
        put("hello, "^get());
        print_string (get());
        0
    }
```
Applet Client Example

The applet client

- first gets a reference to the cell applet server using the name server,
- it then creates a user location where the cell applet will be hosted,
- it requests a new cell applet from the server, and then
- it interacts with the cell through its exported put and get methods.
# Distributed Join Calculus

The distributed join calculus extends the join calculus with primitives for locations, mobility and failure detection.

<table>
<thead>
<tr>
<th>Distributed Join Calculus</th>
<th>JoCaml</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>go ⟨a, k⟩</code></td>
<td><code>go a; p</code></td>
<td>Migration</td>
</tr>
<tr>
<td><code>halt ⟨⟩</code></td>
<td><code>halt ()</code></td>
<td>Termination</td>
</tr>
<tr>
<td><code>fail ⟨a, k⟩</code></td>
<td><code>fail a; p</code></td>
<td>Failure detection</td>
</tr>
<tr>
<td><code>a[D : P]</code></td>
<td><code>let loc a d do p</code></td>
<td>Creation</td>
</tr>
</tbody>
</table>
Distributed Join Calculus

Location migration is modeled as a subjective move whereby a process inside location $a$ moves its enclosing location to another location $b$. Location failures and recovery are supported by two new primitives: $halt$ and $fail$.

- Halting a location makes it permanently inert. We denote it by using the $failed$ keyword. Failed sub-locations can make no progress as no reduction or heating/cooling rules apply.
- The $fail$ primitive detects when a location $a$ or any of its parent locations has failed.

See PDCS’s Figure 10.6 (pp.214) for details on JoCaml’s operational semantics for distribution and mobility.
Mobile Reference Cell in JoCaml

A mobile reference cell can be written in JoCaml as follows:

```
def cell initial_loc c0 =
    let loc mobile_cell
        def content(c) & get() = content(c) & reply c to get
            or content(_) & set(c) = content(c) & reply to set
                and migrate(loc) = go loc & reply to migrate
            in begin
                go initial_loc;
                content(c0); 0
            end
    reply get, set, migrate ;;
spawn begin
    Join.Ns.register Join.Ns.here "cell"
        (cell: Join.location 'a Join.chan);
    Join.Site.listen (Unix.ADDR_INET
        (Join.Site.get_local_addr(),12345));
    def x () & y () = reply to x in x;
    0
end;;
```
Mobile Reference Cell in JoCaml

The cell can be used as follows:

```
let cell =
  (Join.Ns.lookup ns "cell" : Join.location 'a Join.chan) in

let loc user
  do {
      let get, put, mig = cell user "world" in
      put("hello, " ^ get());
      print_string (get());
      mig(there);
      put("hello, new world");
      print_string (get());
      0
  }
```

The cell will be created in the client’s location, used, migrated to a new there location, and used in the new location.