IN.5022 — Concurrent and Distributed Computing

Representing Distributed Algorithms

Prof. P. Felber

pascal.felber@unine.ch

Agenda

- How can we represent distributed algorithms?
- Abstract syntax and guarded actions
- Ensuring important properties

The representation and most examples are based on [Ghosh: Distributed Systems…]

The importance of a representation

- An abstract notation helps in understanding the basic problems that are typical in distributed algorithms, in particular the three properties we will tackle today
  - Non-determinism
  - Atomicity
  - Scheduling and fairness
- Those notions are not built in common programming languages like Java, C++, etc.
  - Languages such as Erlang (or Elixir) are adapted to express distributed algorithms based on these properties

Syntax and semantics

- We will introduce the syntax and semantics for an abstract distributed algorithm representation
  - Many alternative, we will use the one from [Gosh] (simplified)
- Example of headers and simple assignments

```
program name                                  % name of program

define var₁, var₂, var₃: integer              % definitions

initially var₁ = 0                            % initialisation

var₁ := 0                                     % assignments
var₂, var₃ := 1, 2                            % (compound…
var₁, var₂ := var₂, var₁                      % …and swap)
```

Guards

- Sometimes, an action A should occur only when some condition G holds
  - G is called a guard
  - A is a guarded action

```
<guard G> → <action A>                                  % guard
```

- This is somewhat equivalent to a conditional statement

```
if G then A                % "equivalent" conditional statement
```

Alternative constructs

- Multiple guards can be combined in an alternative construct
  - Action A_i occurs only when guard G_i is true
  - If several guards are true, the choice of action is arbitrary
  - If no guard is true, then nothing is done *(skip)*
- *Alternatives can be used to express non-determinism in distributed programs*

```
if                                    % alternative
» G₀ → A₀                                 % guards…
» G₁ → A₁
  …
» Gₙ → Aₙ
fi
```

Repetitive constructs

- Loops can be expressed with a repetitive construct
  - Action A_i occurs only when guard G_i is true
  - If several guards are true, the choice of action is arbitrary
  - The loop is repeated as long as *at least one guard is true*
  - The loop ends *when all guards are false*
- *Repetitions also support non-determinism*

```
do                                        % repetition
» G₀ → A₀                                    % guards…
» G₁ → A₁
  …
» Gₙ → Aₙ
od
```

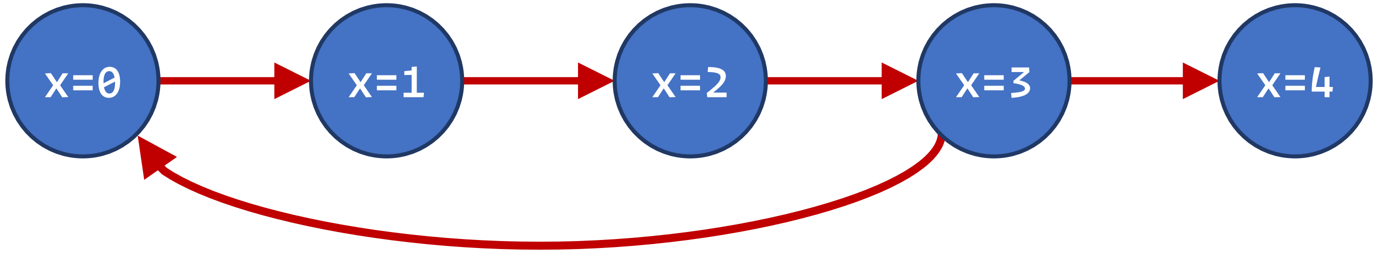
Example: non-determinism

- What happens when this program executes?
  - Will it always terminate?
  - Why?

```
program uncertain

define x: integer

initially x = 0

do
» x < 4 → x := x + 1
» x = 3 → x := 0
od
```

Example: non-determinism

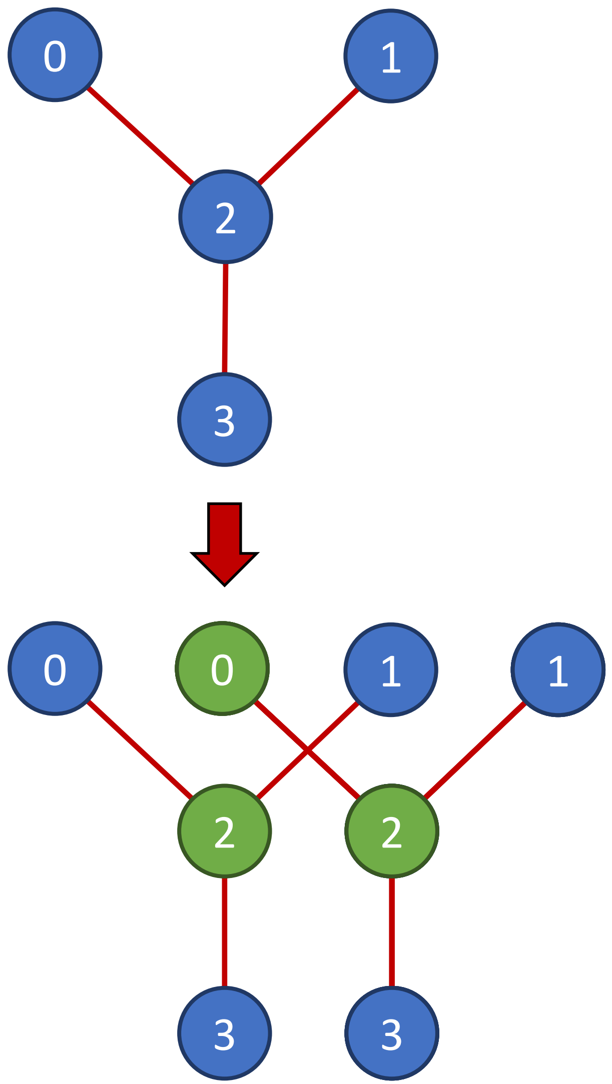
- State-transition diagram is not deterministic
  - Non-deterministic finite automaton



Example: graph colouring

- Four processes and two colours (0, 1)
  - The system must reach a configuration in which no two neighbouring processes have the same colour
  - We assume a central scheduler (one step at a time)
  - Will the program terminate?

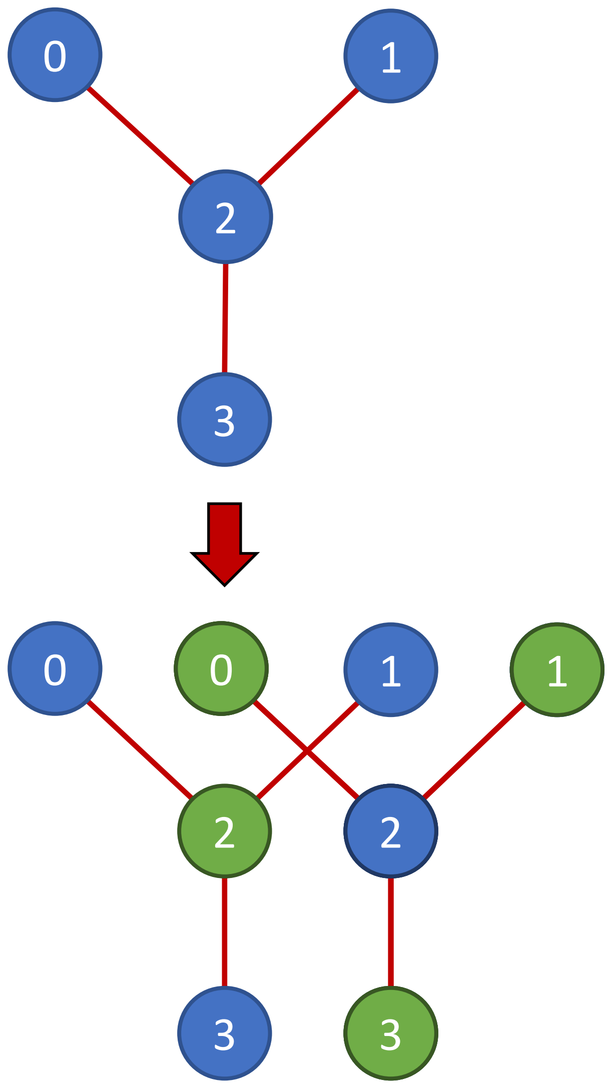
```
program colouring
define c(i): colour
                                % program for process i
do
» ∃j ∈ N(i) : c(i) = c(j) → c(i) := 1 - c(i)
od
```



Example: graph colouring

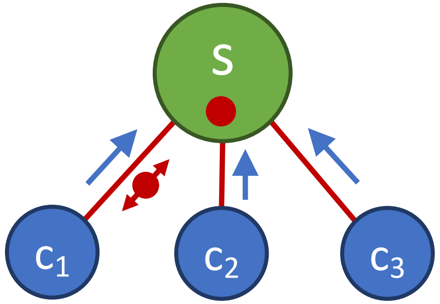
- Let’s change the protocol…
  - Node will change its colour only if it is identical to that of *all* its neighbouring nodes
  - We assume a central scheduler (one step at a time)
  - Will the program terminate?

```
program colouring
define c(i): colour
                          % program for process i
do
» ∀j ∈ N(i) : c(i) = c(j) → c(i) := 1 - c(i)
od
```



Determinism…

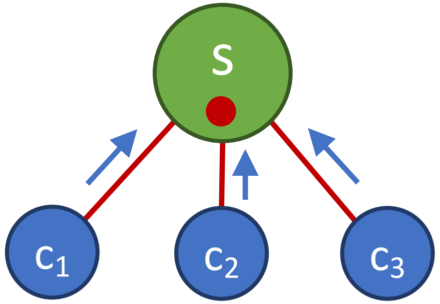
- Token server with 3 clients and 1 token
  - Clients request token, perform some processing and return token to server
  - What's the problem (if any)?




```
program token server₁
do
  if
  » req₁ ∧ token → give token to c₁, wait for token back
  else if
  » req₂ ∧ token → give token to c₂, wait for token back
  else if
  » req₃ ∧ token → give token to c₃, wait for token back
  fi
od
```

…vs. non-determinism

- What if we use non-deterministic choices instead of "if-else"?
  - Deterministic choices are a subset of all possible non-deterministic executions
  - Will this protocol work?



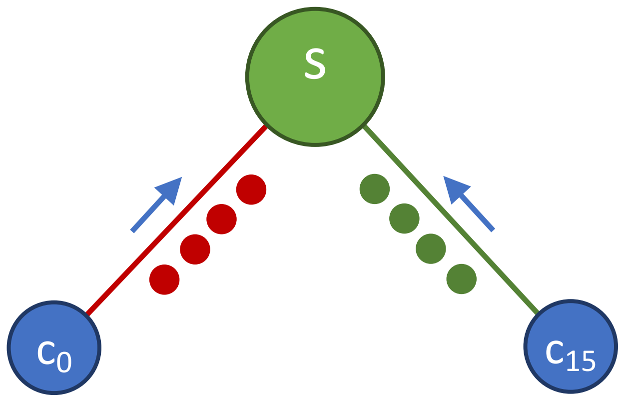

```
program token server₂
do
» req₁ ∧ token → give token to c₁, wait for token back
» req₂ ∧ token → give token to c₂, wait for token back
» req₃ ∧ token → give token to c₃, wait for token back
od
```

Determinism vs. non-determinism

- Should we use deterministic or non-deterministic choices?
  - Deterministic programs have the same behaviour in every run of the program
  - Non-deterministic programs might exhibit a different behaviour at each run, since the scheduler has a discretionary choice about alternative actions
- Both options are good, depending on the problem to be solved
- A system that is proven correct with non-deterministic choices will also be correct with deterministic choices

Atomicity

- Incoming stream of messages (each 4 bits) arriving from 2 channels
  - Red bits (0) and green bits (1)
  - Regardless non-determinism, we would expect that the value of x will be an arbitrary sequence of 0’s and 15’s
  - Is that so?



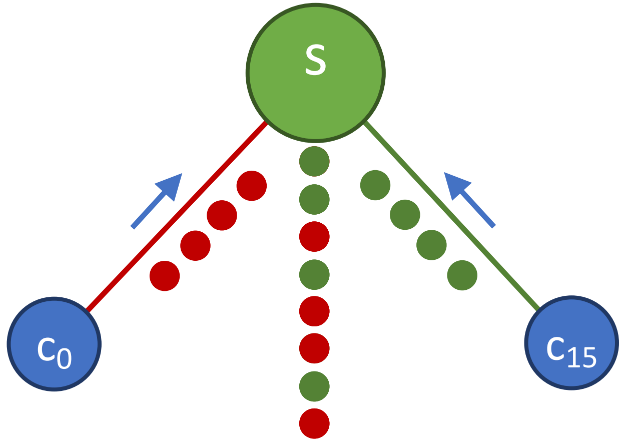

```
program binary
do
» ¬empty(red) → x := 0                          % consume 4 red bits (0000 = 0)
» ¬empty(green) → x := 15                       % consume 4 green bits (1111 = 15)
od
```

Atomicity

- Not necessarily so…
  - Depending on interleaving of red and green bits (and how assignment is done), value could be anything between 0 and 15

    `0101` (5) then `0110` (6)

    `1101` (13) then `0010` (2)…
  - It depends on *atomicity* of assignment
  - Atomicity often require dedicated HW or SW support (RMW operations, transactions, critical sections)
  - We henceforth assume that **G → A** is atomic



Atomicity

- Consider the following program
  - Will it terminate?
  - And what about if we “split” the first guard?

```
program switch
define a, flag: boolean
initially a = true, flag = false

do
» a → flag := true,
      flag := false
» flag AND a → a := false
od
```

```
program switch
define a, flag: boolean
initially a = true, flag = false

do
» a → flag := true
» flag → flag := false
» flag AND a → a := false
od
```

Fairness

- In non-deterministic programs, when multiple guards are true, there are more than one action to choose from
  - The choice of which one to take is determined by the notion of fairness
- Fairness is a property of the scheduler and can affect the behaviour of programs
  - The scheduler is in charge of ordering the activities in a process
  - It makes the ("arbitrary") choices in scheduling activities

Fairness as an adversarial game

- A distributed computation can be viewed as a game between the system and an adversary
  - The adversary may come up with feasible schedules to challenge the system (and cause “bad things”)
  - A correct algorithm must be able to prevent those bad things from happening
- Fairness defines the restrictions on the scheduling of actions
  - No restriction implies an unfair scheduler
- Fair schedulers can provide different levels of fairness
  - Unconditional fairness
  - Weak fairness
  - Strong fairness

Fairness levels

- Consider the following program
  - An unfair scheduler may never schedule the 2nd and 3rd actions and **x** may remain zero
  - An unconditionally fair scheduler eventually schedule each statement, regardless of the value of its guard
  - A weakly fair scheduler eventually schedules every guarded action whose guard becomes true and remains true thereafter (incl. 2nd action but excl. 3rd action)
  - A strongly fair scheduler eventually schedules every guarded action whose guard is true infinitely often (incl. 3rd action)

```
program scheduler
define x: integer
    % initial value unknown
do
» true → x := 0
» x = 0 → x := 1
» x = 1 → x := 2
od
```

Summary

- Non-determinism, atomicity and fairness are important aspects of concurrent and distributed programming
  - The semantics of the computation depend on specific assumptions about atomicity, non-determinism and scheduling policies
- Implementing distributed programs in Java, C++, etc. requires to implement not only the guards, but also the intended grain of atomicity, non-determinism and appropriate fairness of scheduler (not trivial!)
  - Languages like Erlang provide built-in support for those

Erlang syntax

- Erlang maps (quite) closely to abstract syntax
  - Example: two processes exchange a message M times between each other (“ping” and “pong” messages)
  - Graceful termination (via “finished” message)

```
program pong
do
» ping → reply pong
» finished → break
od
```

```
-module(pingpong).
-export([start/0, ping/2, pong/0]).
ping(0, Pong_PID) ->
  Pong_PID ! finished;
ping(N, Pong_PID) ->
  Pong_PID ! {ping, self()},
  receive
    pong -> io:format("Pong!~n", [])
  end,
  ping(N - 1, Pong_PID).
pong() ->                         % function
  receive                         % receive message
    {ping, Ping_PID} ->           % guard
      io:format("Ping!~n", []),
      Ping_PID ! pong,            % send message
      pong()
    finished -> true;
  end.
start() ->                        % spawn both processes
  Pong_PID = spawn(?MODULE, pong, []),
  spawn(?MODULE , ping, [3, Pong_PID]).
```

Elixir syntax

- *Elixir* maps (quite) closely to abstract syntax
  - Example: two processes exchange a message M times between each other ("ping" and "pong" messages)
  - Graceful termination (via "finished" message)

```
program pong
do
» ping → reply pong
» finished → break
od
```

```
defmodule PingPong do
  def ping(0, pong_pid) do
    send(pong_pid, :finished)
  end
  def ping(n, pong_pid) do
    send(pong_pid, {:ping, self()})
    receive do
      :pong -> IO.puts("Pong!")
    end
    ping(n - 1, pong_pid)
  end
  def pong do                          % function
    receive do                  % receive message
      {:ping, ping_pid} ->             % guard
        IO.puts("Ping!")
        send(ping_pid, :pong)
        pong()
      :finished -> true
    end
  end
  def start do        % spawn both processes
    pong_pid = spawn(fn -> pong() end)
    spawn(fn -> ping(3, pong_pid) end)
  end
end
```