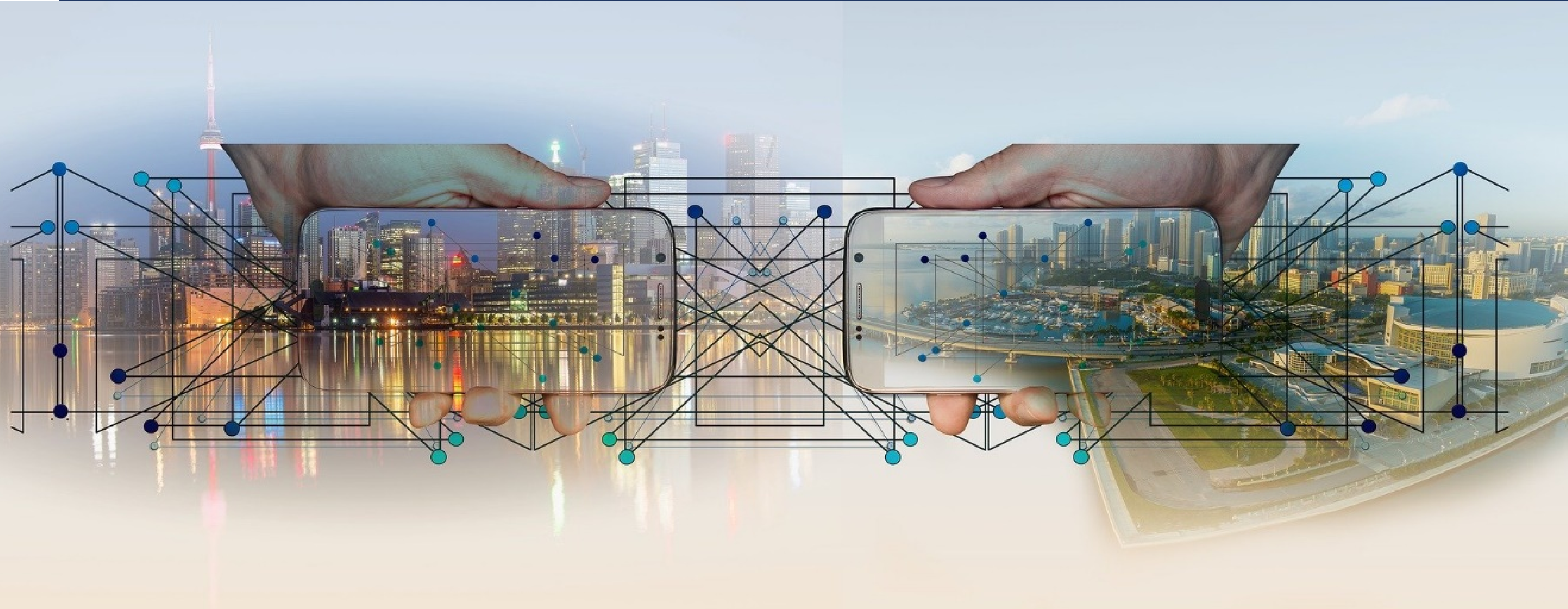
Lecture 8

Representing Distributed Algorithms



Representing distributed algorithms

Why do we need these?
Don't we already know
a lot about programming?

Well, you need to capture the notions of *atomicity, non-determinism, fairness* etc.
These concepts are not built into languages like JAVA, C++ etc!

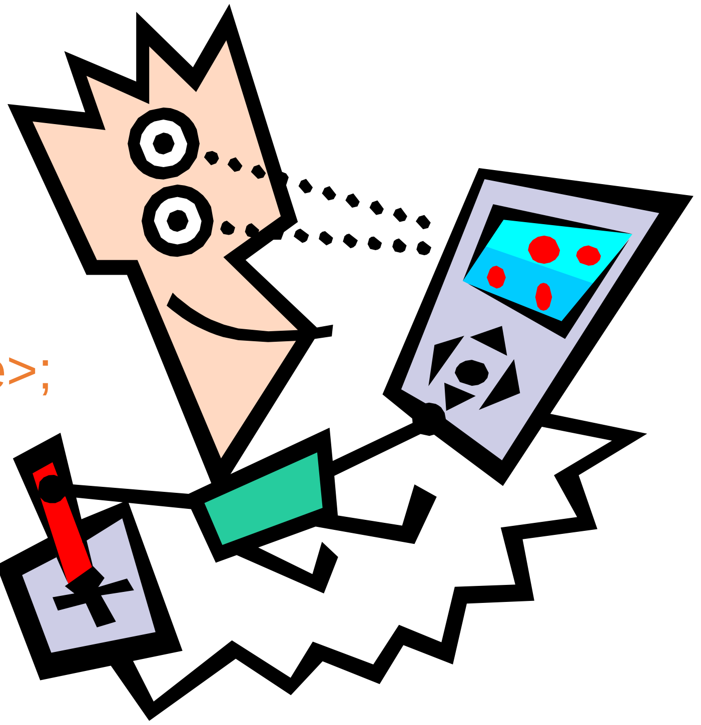
Syntax & semantics

- Structure of a program
  - In the opening line

    program <name>;
  - To define variables and constants

    define <variable name>: <variable type>;

```
(ex 1) define n: message;
(ex 2) type message = record
                   a: integer
                   b: integer
                   c: boolean
                   end
         define    m: message
```



Syntax & semantics

- To assign <u>an initial value</u> to a variable

  Initially <variable> = <initial value>;

  (ex) initially x = 0;

- A simple assignment

  <variable> := <expression>

  (ex) x := E

- A compound assignment

  <variable 1>[,<variable n>] :=
  <expression 1>[,<expression n>]

  (ex) x, y := m.a, 2

  It is equivalent to x := m.a and y := 2

Syntax & semantics

- Example (We will revisit this program later.)

```
program uncertain;
define x : integer;
initially x = 0;
do x < 4 → x := x + 1
□ x = 3 → x := 0
od
```

Syntax & semantics

- Guarded Action: Conditional Statement

  <guard G> $\rightarrow$ <action A>

  is equivalent to

  if G then A

- Not: $\neg$

Syntax & semantics

- *Sequential actions* S0; S1; S2; . . . ; Sn
- *Alternative constructs* **if** . . . . **fi**
- *Repetitive constructs* **do** . . . . **od**

The specification is useful for representing abstract algorithms, not executable codes.

Syntax & semantics

Alternative construct

```
if      $G_1 \rightarrow S_1$
□       $G_2 \rightarrow S_2$
...
□       $G_n \rightarrow S_n$
fi
```

When no guard is true, **skip** (do nothing). When multiple guards are true, the choice of the action to be executed is **completely arbitrary**.

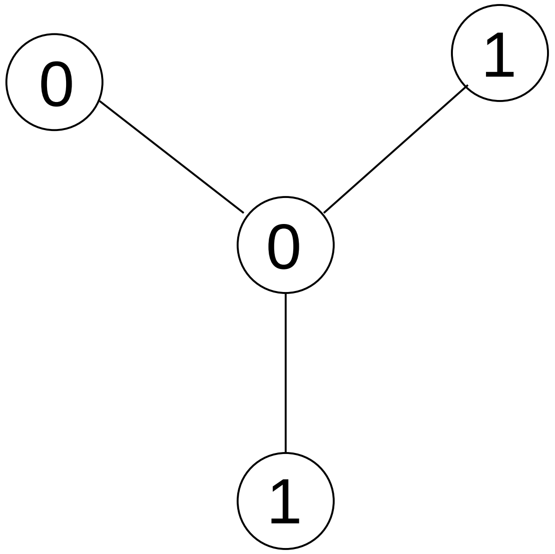
Syntax & semantics

Repetitive construct

```
do    $G_1 \rightarrow S_1$
□     $G_2 \rightarrow S_2$
.
□     $G_n \rightarrow S_n$
od
```

Keep executing the actions until *all guards* are false and the program terminates. When multiple guards are true, the **choice of the action is arbitrary.**

Example: graph coloring



There are four processes. The system has to reach a configuration in which no two neighboring processes have the same color.

```
{program for process i}
do
$\exists j \in$ neighbor(i): c(j) = c(i) $\rightarrow$ c(i) := 1-c(i)
od
```

Will the above computation terminate?

Consider another example

```
program    uncertain;
define     x  :   integer;
initially  x = 0
do     x < 4 → x := x + 1
□      x = 3   →   x := 0
od
```

Question. Will the program terminate?

- Our goal here is to understand **fairness**
- A Major issue in a distributed computation is **global termination**

The adversary

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.

Deterministic Computation vs. Nondeterministic Comp.

- Deterministic Computation
  - The behaviors remains the same during every run of the program
- Nondeterministic Computation
  - The behaviors of a program may be different during different runs since the scheduler may choose other alternative actions.

Non-determinism

```
define x: array [0..k-1] of boolean
initially all channels are empty
do ¬ empty ($c_0$) → send ACK along $c_0$
□ ¬ empty ($c_1$) → send ACK along $c_1$
□            …
□ ¬ empty ($c_{k-1}$) → send ACK along $c_{k-1}$
od
```

Is it fair?

➔For example, if all three requests are sent simultaneously, client 2 or 3 may never get the token with a deterministic scheduler! The outcome could have been different if the server makes a non-deterministic choice

Non-determinism is abundant in the real world. Examples?

Examples of non-determinism

- Non-determinism is abundant in the real world.
  - If there are multiple processes ready to execute actions, who will execute the action first is nondeterministic.
  - If message propagation delays are arbitrary, the order of message reception is non-deterministic

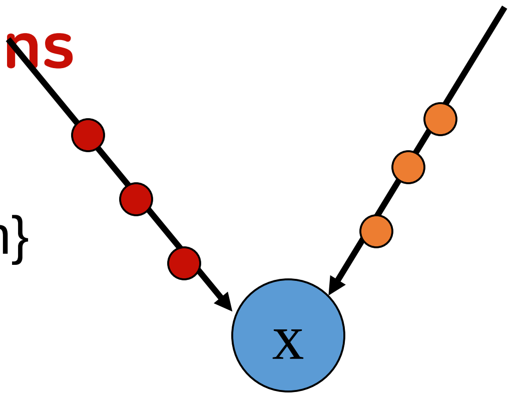
Determinism has a specific order and is a special case of non-determinism.

Atomicity (or granularity) (1)

Atomic = all or nothing

Atomic actions = indivisible actions

```
do      red message → x:= 0   {red action}
□  blue message → x:=7    {blue action}
od
```



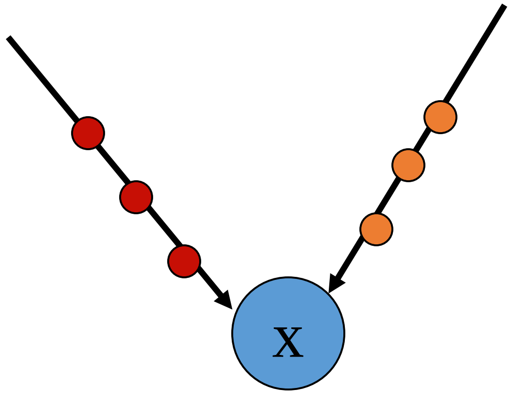
Regardless of how nondeterminism is handled, we would expect that the value of x will be an arbitrary sequence of 0 and 7.

Right or wrong?

Atomicity (2): [Q] Assignment?

```
do    red message → x:= 0   {red action}
□  blue message → x:=7    {blue action}
od
```

Let x be a 3-bit integer x2 x1 x0, so
x:=7 means x2:=1, x1:= 1, x2:=1, and
x:=0 means x2:=0, x1:= 0, x2:=0



If the assignment is not atomic, then many Interleavings are possible, leading to any possible values of x

So, the answer may depend on atomicity

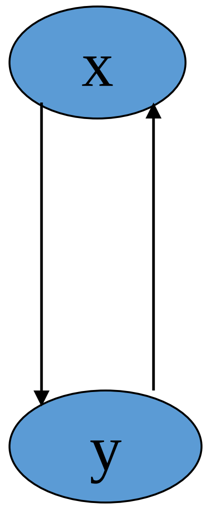
Atomicity (4): [Q] Critical Section Code?

Unless stated, we will assume that $G \rightarrow A$ is an "atomic operation." Does it make a difference if it is not so?

Transactions are atomic by definition (in spite of process failures). Also, critical section codes are atomic.

Can the other process B read the state of the process A while the process A is executing the if statement?

if $x \neq y \rightarrow x := \neg x$ **fi**



if $x \neq y \rightarrow y := \neg y$ **fi**

Fairness

Defines the choices or restrictions on the scheduling of actions. No such restriction implies an unfair scheduler. For fair schedulers, the following types of fairness have received attention:

- Unconditional fairness
- Weak fairness
- Strong fairness

Scheduler / demon / adversary

Fairness

```
Program      test
define       x  : integer
{initial value unknown}
do      true   →   x : = 0
□       x = 0  →   x : = 1
□       x = 1  →   x : = 2
od
```

An unfair scheduler *may never* schedule the second (or the third actions). So, x may always be equal to zero.

An unconditionally fair scheduler will *eventually* give every statement a chance to execute without checking their eligibility. (Example: process scheduler in a multiprogrammed OS.)

Weak fairness

```
Program test
define          x  : integer
{initial value unknown}
do      true   →   x : = 0
□       x = 0  →   x : = 1
□       x = 1  →   x : = 2
od
```

- A scheduler is weakly fair, when it eventually executes every guarded action whose guard becomes true, and remains true thereafter

- A weakly fair scheduler will eventually execute the second action, but may never execute the third action. Why?

Strong fairness

```
Program     test
define      x  : integer
{initial value unknown}
do      true   →   x : = 0
□       x = 0  →   x : = 1
□       x = 1  →   x : = 2
od
```

- A scheduler is strongly fair, when it eventually executes every guarded action whose guard is true infinitely often.

- The third statement will be executed under a strongly fair scheduler. Why?

Central vs. Distributed Scheduler

- Distributed Scheduler
  - Since each individual process has a local scheduler, it leaves the scheduling decision to these individual schedulers, without attempting any kind of global coordination.

- Central Scheduler or Serial Scheduler
  - It based on the interleaving of actions. It assumes that an invisible demon finds out all the guards that are enabled, arbitrarily picks any one of these guards, schedules the corresponding actions, and waits for the completion of this action before re-evaluating the guards.

Example: Central vs. Distributed Scheduler

- Goal: To make x[i+1 mod 2] = x[i]

```
{in all the processors i}
do
□  x[i+1 mod 2] ≠ x[i] → x [i] := ¬ x[i]
od
```

Will this program terminate?

- using distributed scheduler
- using central scheduler

Simulation of a Distributed scheduling model

- Example
  - Let y[k,i] denotes the local copy of the state x[k] of process k as maintained by a neighboring process i.
    - To evaluate the guard by process i
      - process i copies the state of each neighbor k, that is, y[k,i] := x[k]
      - Each process evaluates its guard(s) using the local copies of its neighbors' state and decides if an action will be scheduled.
    - The number of steps allowed to copy the neighbors' states will depend on the grain of atomicity.
      - Read-write atomicity in a fine-grain atomicity: only one read at a time
      - Coarse-grain atomicity model: all the read can be done in a single step

Advantage & Disadvantage of Central scheduling

- Advantage
  - Relatively easy of correctness proof
- Disadvantage
  - Poor parallelism and poor scalability

➔ To avoid a serious problem, a correctness proof of the designed scheduler (scheduling algorithm) is very important.

Example: Correctness proof (1)

- No System function correctly with distributed schedulers unless it functions correctly under a central scheduler.
- In restricted cases, correct behavior with a central scheduler guarantees correct behavior with a distributed scheduler.
  - Theorem 4.1 If a distributed system works correctly with a central scheduler and no enabled guard of a process is disabled by the actions of their neighbors, the system is also correct with a distributed scheduler.

Example: Correctness proof (2)

- Proof. Assume that i and j are neighboring processes. Consider the following four events: (1) the evaluation of G_i as true; (2) the execution of S_i; (3) the evaluation of G_j as true; and (4) the execution of S_j. Distributed schedulers allow the following schedules:

  - Case 1: (1)(2)(3)(4)
  - Case 2: (1)(3)(4)(2)
  - Case 3: (1)(3)(2)(4)

  Since the case 2 and the case 3 can be reduced to the case 1 and the case 1 corresponds to that of a central schedule. Thus, the theorem is proven.