Message Passing Concurrency

Systemes Concurrents

TD 3

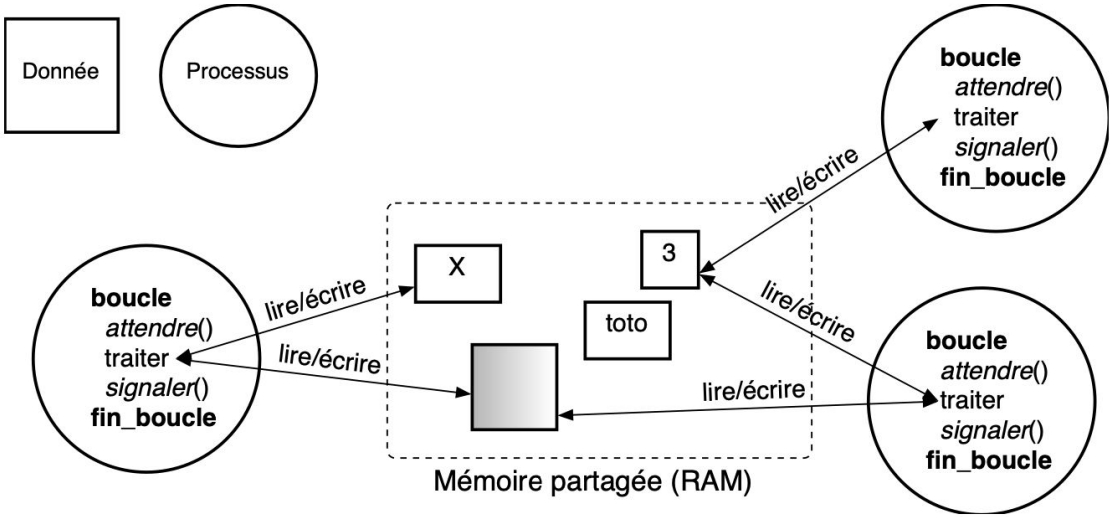
Simone Gasparini
simone.gasparini@toulouse-inp.fr

Outline

- Message Passing Concurrency
- Trains on a single track
- Bridge tournament

Previously...

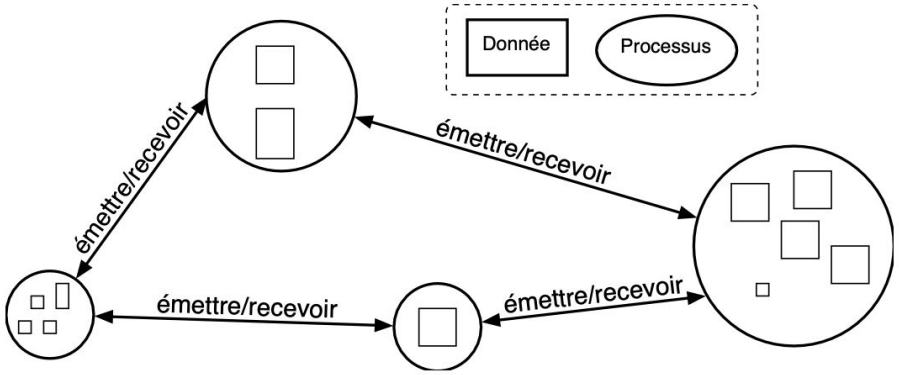
Modèles d'interaction : mémoire partagée



- Données partagées
- Communication implicite
  - résulte de l'accès et de la manipulation des variables partagées
  - l'identité des activités n'intervient pas dans l'interaction
- Synchronisation explicite (et nécessaire)
- Architectures/modèles cibles
  - multiprocesseurs à mémoire partagée,
  - programmes multiactivités

Message Passing Concurrency

Modèles d'interaction : processus communicants



- Données encapsulées par les processus
- Communication nécessaire, explicite : échange de messages
  - Programmation et interactions plus lourdes
  - Visibilité des interactions → possibilité de trace/supervision
  - Isolation des données
- Synchronisation implicite : attente de message
- Architectures/modèles cibles
  - systèmes répartis : sites distants, reliés par un réseau
  - moniteurs, CSP/Erlang/Go, tâches Ada

Message Passing Concurrency

Processus communicants

Principes

- Communication inter-processus avec des opérations explicites d'envoi / réception de messages
- Synchronisation via ces primitives de communication bloquantes : envoi (bloquant) de messages / réception bloquante de messages

Communicating Sequential Processes (CSP) / Calculus of Communicating Systems (CCS) / π-calcul / Erlang / Go

Message Passing Concurrency

Activité gestionnaire d'un objet partagé

Interactions avec l'objet partagé

Pour chaque opération faisable sur l'objet :

- émettre un message de requête vers le gestionnaire
- attendre le message de réponse de gestionnaire

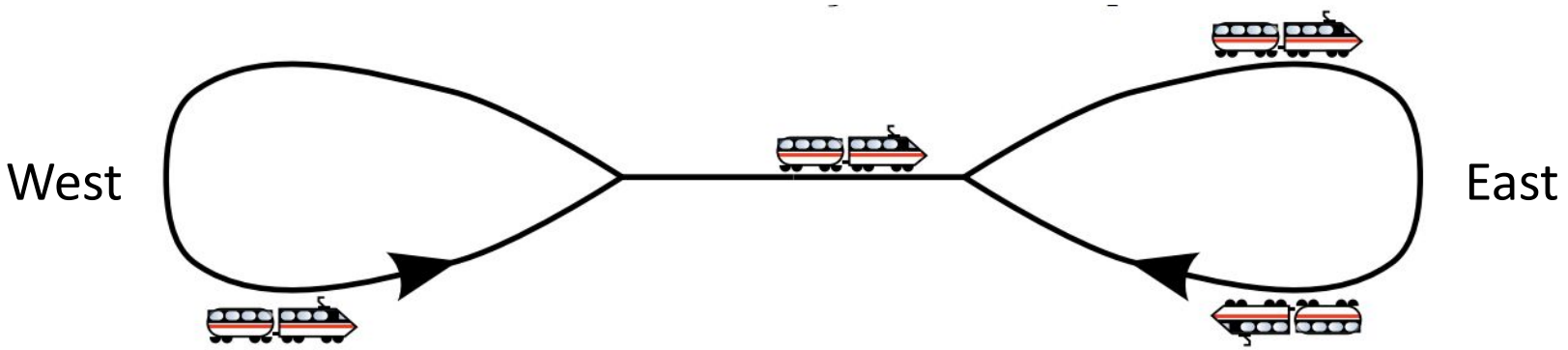
Schéma de fonctionnement du gestionnaire

- L'arbitre exécute une boucle infinie contenant une alternative
- Cette alternative possède une branche par opération fournie
- Chaque branche est gardée par la condition d'acceptation de l'opération (suivie de l'attente du message correspondant)

Note : en communication synchrone, on peut se passer du message de réponse s'il n'y a pas de contenu à fournir.

Single track problem

Problem statement



- Single track section
  - only trains going in the same direction (WE or EW) at the same time

Solve synchronization problem with

1. acceptance conditions
2. a finite-state machine

Synchronisation pure, approche condition

Pour un problème de synchronisation pure (pas d'échange de données) :

1. Identifier l'interface = les requêtes recevables $\rightarrow$ un canal par requête
2. Identifier les conditions d'acceptation pour chaque canal
3. Activité serveur qui boucle, lisant un message parmi ceux dont la condition d'acceptation est vraie (et bloquant s'il n'y a aucun tel message).

Methodology

1. Define the **interfaces**
   a. identify the communication channels
2. For each channel state the **acceptation conditions**
3. Define the **variables** needed to express the conditions
4. Implement the **synchronization**

Define the interfaces

- Define the needed channels
- Here we have to communicate
  - the intention to enter the single track for one of the two directions
  - the exit of the single track

Hence:

- `enterEW  // entering the single track in EW direction`
- `enterWE`
- `exitST   // exiting the single track`

The code of a train would be like:

```
loop
    enterEW.send()
    exitST.send()
    // change direction
    enterWE.send()
end
```

Acceptation conditions

For each channel specify the condition for receiving a message

Channel	Condition
`enterEW`	the single track is either empty or there are trains going in the same direction EW
`enterWE`	the single track is either empty or there are trains going in the same direction WE
`exitST`	nothing

Variables

Variables necessary to express the conditions

`direction : (EW, WE)`	the direction of the train(s) currently in the single track
`nbTrains : int`	number of trains in the single track

Acceptation conditions

For each channel specify the condition for receiving a message

Channel	**Condition**
`enterEW`	`direction == EW OR nbTrains == 0` *the single track is either empty or there are trains going in the same direction EW*
`enterWE`	`direction == WE OR nbTrains == 0` *the single track is either empty or there are trains going in the same direction WE*
`exitST`	nothing

CSP syntax

CSP

```
*[
    (cond) --> channel1?var1;
  □
    (cond) --> channel2?_;
  □
     channel3?var3
]
```

Pseudo code (hopefully more readable)

```
loop
   select
    case (cond)
        var1 = channel1.receive()

    case (cond)
        channel2.receive()

    case:  // no condition
        var3 = channel3.receive()
end
```

Implementation

```
variables
direction : (EW, WE)
nbTrains : integer
loop
    select
        case ( nbTrains = 0 OR direction = EW )
            enterEW.receive()
            nbTrains++
            direction:=EW

        case ( nbTrains = 0 OR direction = WE )
            enterEW.receive()
            nbTrains++
            direction:=WE

        case:
            exitST.receive();
            nbTrains--;
    end
end
```

Remarks

- This solution can lead to starvation as it does not assure **fairness**
- To implement fairness we need to split the enter channels into 2 different channels
  - 1 to manifest the intention to go in a given direction
  - 1 as before to access the single track in that given direction
- See next exercise

Implementation in GO

```
const (dirEW = iota
       dirWE
)
/// simulates the code of a train entering the single track in given direction
func train(msg string, enter chan bool, exit chan bool) {
    for {
        enter <- true
        fmt.Println("In :  ", msg)
        time.Sleep(time.Duration(rand.Int31n(1000)) * time.Millisecond)
        fmt.Println("Out : ", msg)
        exit <- true
        time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
    }
}
func main() {
    enterEW := make(chan bool)
    enterWE := make(chan bool)
    exit := make(chan bool)

     // start the track controller and the trains
    go singleTrack(enterEW, enterWE, exit)
    for i := 1; i < 5; i++ {
        go train("WE", enterWE, exit)
        go train("EW", enterEW, exit)
    }
    time.Sleep(20 * time.Second)
    fmt.Println("DONE. ")
}
```

Test it on Go playground https://go.dev/play/p/U2znKWQ7GoV

Implementation in GO

```
const (dirEW = iota
       dirWE
)
// when(b, c) returns c if b is true, nil otherwise
func when(b bool, c chan bool) chan bool {
    if b {
        return c
    } else {
        return nil
    }
}
// singleTrack simulates a single track where trains can go in both directions but not at the same time
func singleTrack(enterEW chan bool, enterWE chan bool, exit chan bool) {
    nbTrains := 0
    direction := dirEW
    for {
        select {
              case <- when(nbTrains == 0 || direction == dirEW, enterEW):
                  nbTrains++
                  direction = dirEW
              case <- when(nbTrains == 0 || direction == dirWE, enterWE):
                  nbTrains++
                  direction = dirWE
              case <- exit:
                  nbTrains--
                  if nbTrains == 0 {
                      fmt.Println("*** : Track available.")
            }
        }
    }
}
```

Synchronisation pure, approche par automate

Pour un problème de synchronisation pure (pas d'échange de données) :

1. Identifier l'interface = les requêtes recevables $\rightarrow$ un canal par requête
2. Construire un automate fini à états.
   Chaque état se distingue par les canaux acceptables en lecture.
3. L'activité serveur boucle, et selon l'état courant détermine quels messages peuvent être pris.

Synchronization with finite-state machine

- Define the states
- Define the variables
  - they are used for the state transition
- The channels remain the same

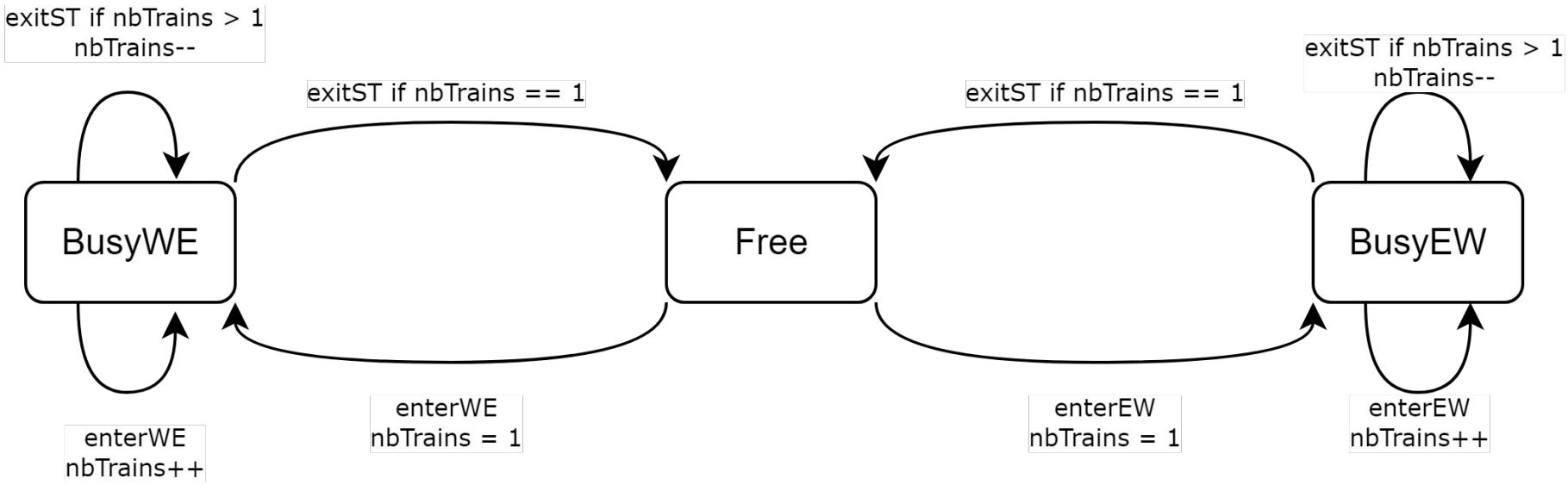
States

- Free
- BusyWE
- BusyEW

Variables

- nbTrains

State machine



Implementation

```
variables
state : (Free, busyEW, busyWE)
nbTrains : integer
loop
     switch(state)
          Free:
            select
                case:
                    enterEW.receive(), nbTrains = 1, state=BusyEW
                case:
                    enterWE.receive(), nbTrains = 1, state=BusyWE
          busyEW:
            select
                case:
                    enterEW.receive(), nbTrains++, state=BusyEW
                case:
                    exitST.receive(), nbTrains--,
                    if nbTrains == 0 then state = Free
          busyWE:
            select
                case:
                    enterWE.receive(), nbTrains++, state=BusyWE
                case:
                    exitST.receive(), nbTrains--,
                    if nbTrains == 0 then state = Free

end
```

Implementation in GO

```
func voieUnique(enterEW chan bool, enterWE chan bool, exitST chan bool) {
    nbTrains := 0
    state := Free
    for {
        if state == Free {
            select {
                case <- enterEW:
                    nbTrains = 1
                    state = dirEW
                case <- enterWE:
                    nbTrains = 1
                    state = dirWE
            }
        } else if state == dirEW {
            select {
                case <- enterEW:
                    nbTrains++
                case <- exitST:
                    nbTrains--
                    if nbTrains == 0 {
                        state = Free
                    }
            }
        } else if state == dirWE {
            select {
                case <- enterWE:
                    nbTrains++
                case <- exitST:
                    nbTrains--
                    if nbTrains == 0 {
                        state = Free
                    }
            }
        }
    }
    }
```

Bridge Tournament

Modeling a bridge tournament to ensure that the number of people present in the room is always a multiple of 4.

Constraints:

- a player can **enter** if another is ready to leave (exchange)
  or
  if there are already **three other players waiting** to enter (the group of four can then enter);

- one player can **exit** if another is ready to enter (exchange)
  or
  if there are already **three other players waiting** to exit (the group of four can then exit).

Note: we're not interested in assigning players to tables.

Solution with acceptance conditions

- The rendez-vous can be accepted as a single or a block of 3 others
- Need to split the each action (enter, exit) into 2 parts
  - **announcement** of intent
  - then the blocking request itself.

Define the interfaces

- Define the needed channels
- Here we have to communicate
  - the intention to enter or exit (always accepted)
  - entering or exiting the room (blocking)

Hence:

```
• announceEnter  // willing to enter
• enter          // enter the room
• announceExit   // willing to exit the room
• exit           // exit the room
```

The code of a player would be like:

```
loop
    announceEnter.send()
    enter.send()
    // play some bridge
    announceExit.send()
    exit.send()
end
```

Acceptation conditions

For each channel specify the condition for receiving a message

Channel	Condition
`announceEnter`	nothing, always accept
`enter`	if there is at least 1 willing to exit and one willing to enter OR if there is at least 4 willing to enter
`announceExit`	nothing, always accept
`exit`	if there is at least 1 willing to exit and one willing to enter OR if there is at least 4 willing to exit

Variables

Variables necessary to express the conditions, we need to count the players that are willing to enter or exit

`nbEnter : int`	the number of players that are willing to enter
`nbExit : int`	the number of people that are willing to exit

Acceptation conditions

For each channel specify the condition for receiving a message

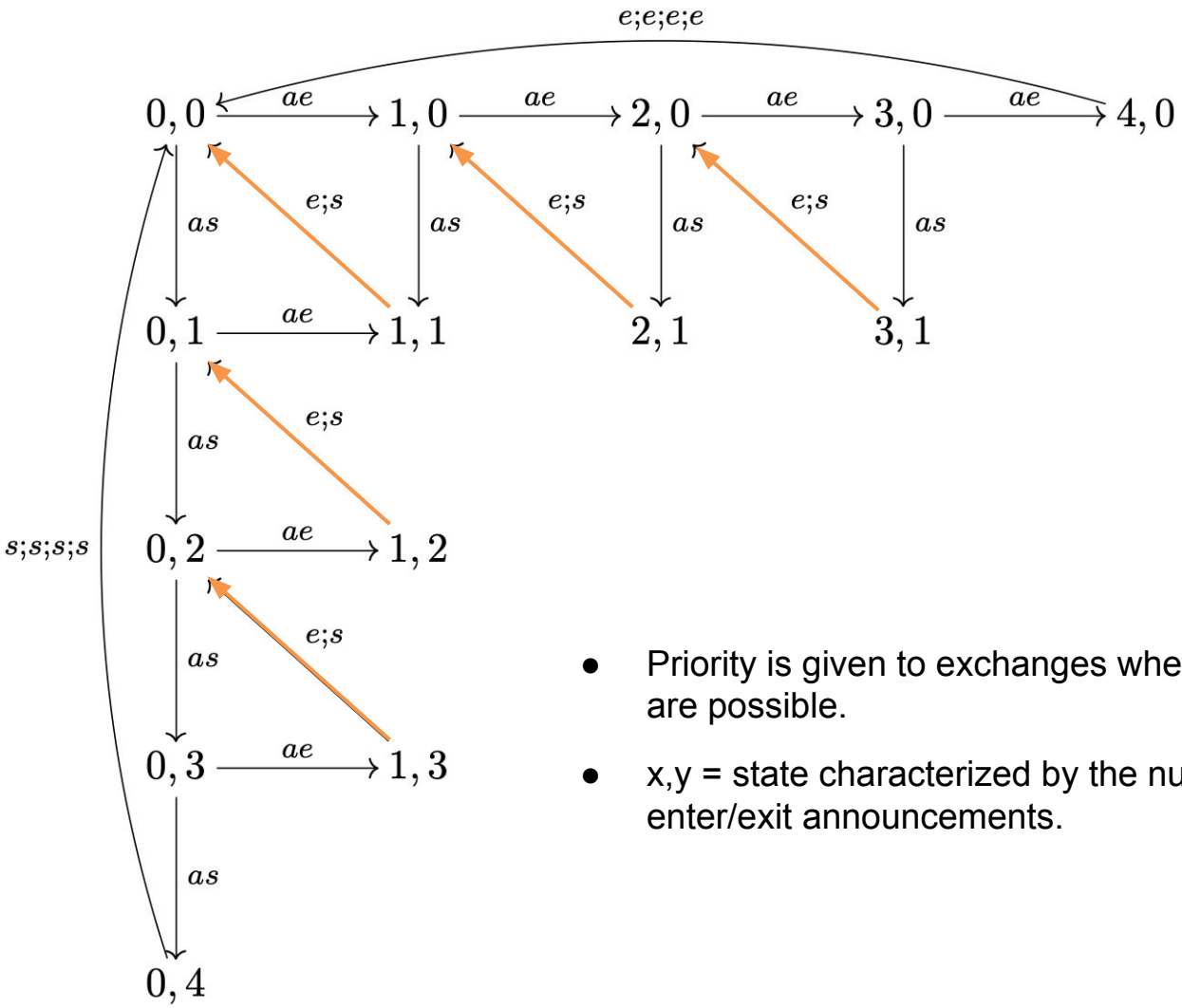
Channel	Condition
`announceEnter`	nothing, always accept
`enter`	`(nbEnter ≥ 1 AND nbExit ≥ 1) OR (nbEnter ≥ 4)` *if there is at least 1 willing to exit and one willing to enter* *OR* *if there is at least 4 willing to enter*
`announceExit`	nothing, always accept
`exit`	`(nbEnter ≥ 1 AND nbExit ≥ 1) OR (nbExit ≥ 4)` *if there is at least 1 willing to exit and one willing to enter* *OR* *if there is at least 4 willing to exit*

Implementation

```
variables
nbEnter, nbExit : integer
loop
    select
        case:
            announceEnter.receive(), nbEnter++

        case:
            announceExit.receive(), nbExit++

        case (nbEnter ≥ 1 AND nbExit ≥ 1)
            enter.receive(), exit.receive()
            nbEnter--, nbExit--

        case (nbEnter ≥ 4)
            enter.receive(), enter.receive(),
            enter.receive(), enter.receive()
            nbEnter -= 4

        case (nbExit ≥ 4)
            enter.receive(), exit.receive(),
            exit.receive(), exit.receive()
            nbExit -= 4
    end
end
```

State machine version

ae / as = annonce entrée / sortie ; e / s = entrer / sortir.



- Priority is given to exchanges whenever they are possible.
- x,y = state characterized by the number of enter/exit announcements.