

# Algorithmes et Systèmes Distribués : TP1

E. Fabiani

## Environnement des TP Go

Pour les TP on utilise le compilateur go installé en local sur les machines sous windows ou linux (en cas de problème, on peut se rabattre sur le playground, voir plus loin). L'utilisation basique est :

- Créer un répertoire (par exemple nommé *go*) sur votre compte (dans H :), avec des sous-répertoires (par exemple *tp1*)
- Éditez vos programmes (par exemple *helloWorld.go*) dans ce répertoire
- Utilisez un terminal windows ou linux (invite de commande) pour compiler :
  - Dans le cas de windows :
    - pour lancer le terminal sous windows : raccourci clavier (logo windows + R) pour afficher le lanceur d'applications, puis entrer *cmd*
    - pour aller sur votre compte : taper H : suivi de *entrée*
    - pour aller dans votre répertoire : *cd go/tp1*
    - pour lister le contenu du répertoire : *dir* (au lieu du *ls* de linux)
    - pour compiler un programme : *go build helloWorld.go*
    - pour l'exécuter : *helloWorld*
  - Dans le cas de linux, par exemple :
    - Aller sur votre répertoire de TP dans le navigateur de fichier et sélectionner "ouvrir dans un terminal" via un click droit
    - pour lister le contenu du répertoire : *ls*
    - pour compiler un programme : *go build helloWorld.go*
    - pour l'exécuter : *./helloWorld*
- Dans le cas d'un programme sans fin (boucle infinie), arrêter l'exécution avec *control+C*
- Si vous voulez stocker la trace exécution (affichages écran du programme) pour l'analyser, utiliser la redirection. Par exemple : *helloWorld > trace.txt*

Vous êtes libre d'utiliser l'éditeur de texte que vous souhaitez (sauf le bloc-note !), notePad++ est disponible sous windows. Vous êtes libre également d'utiliser un IDE pour ne pas passer par un terminal, mais ce n'est pas indispensable.

**Cas du playground** En cas de problème (compilateur non présent sur la machine, ...) on peut aussi utiliser le "playground" de go (<https://go.dev/play/>). Il s'agit d'une page web permettant de saisir un programme go, de le compiler et de l'exécuter via un serveur distant. Cet environnement de programmation a l'avantage d'être portable et immédiatement disponible mais il a évidemment des défauts :

- Pour la conservation des programmes, il faut penser à les recopier dans des fichiers locaux
- Il n'y a pas d'accès à des fichiers locaux (pour les entrées/sorties)
- Par défaut, le programme ne s'exécute que sur un seul cœur (sur le serveur distant)
- L'affichage n'apparaît pas instantanément : l'exécution est réalisée sur le serveur distant, la trace d'affichage est ensuite renvoyée d'un seul bloc sur le client local. Ce n'est donc pas adapté pour les programmes avec boucle infinie.

## Sources de documentation

- La page du langage go est <https://go.dev> (pour installation et documentation).
- Un tutoriel (utilisant le playground) est disponible sur <https://go.dev/tour> . Il peut permettre d'étendre et de compléter la connaissance des éléments de base vus en cours (à faire chez soi).
- Si vous cherchez une documentation ou un exemple sur un problème précis sur internet, utilisez bien *golang* comme mot clé (le mot clé *go* donnera beaucoup de réponses non pertinentes).



## 1 Hello world, goroutine et interblocage

1. Éditez, compilez et testez le programme suivant :

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello ")
    fmt.Println("world ")
}
```

2. Ajoutez une goroutine incluant l’affichage du “Hello” en utilisant une fonction anonyme (sans synchronisation) et testez le comportement sur plusieurs exécutions
3. Ajoutez une synchronisation via un canal, afin que l’affichage soit conforme au programme initial
4. Enlevez l’envoi dans le canal, testez et observez le signallement de l’interblocage

## 2 Définition d’un pipeline avec processus communicants

Il s’agit de se familiariser avec la programmation en Go, en introduisant progressivement des éléments de syntaxe nouveaux en se basant sur une application commune (seuillage de valeurs de pixels) que l’on va faire évoluer incrémentalement. Il est conseillé de sauvegarder le programme à chaque question afin de garder un historique de la résolution des questions et pouvoir revenir en arrière en cas d’erreur.

### 2.1 Création du système de départ (simpliste)

Soit 4 processus *source*, *seuillage*, *affichage* et *main* dont les comportements sont décrits dans les parties suivantes.

#### 2.1.1 Méthodologie

1. Schématisez le système, en annotant les noms des processus, des canaux, et les types des canaux
2. Écrivez le *main* en vous basant sur le schéma
3. Écrivez les fonctions *source*, *seuillage*, *affichage*, compilez et testez

#### 2.1.2 Processus *main*

Le processus *main* lance les goroutines *source*, *seuillage*, *affichage* en parallèle, connecte la sortie de *source* à *seuillage* et la sortie de *seuillage* à *affichage*. Il est nécessaire de synchroniser la fin du *main* avec la fin des goroutines, mais sans que cela modifie le code de *source*, *seuillage*, et *affichage*.

#### 2.1.3 Processus *source*

Le processus *source* envoie un entier sur un canal en sortie. La valeur de l’entier est fixe.

#### 2.1.4 Processus *seuillage*

Le processus *seuillage* a pour paramètre la valeur du seuil (entier compris entre 0 et 255), un canal d’entrée et un canal de sortie.

Le processus reçoit sur son canal d’entrée la valeur d’un pixel et envoie sur son canal de sortie un entier égal à 0 si le pixel est inférieur au seuil, ou égal à 1 si le pixel est supérieur ou égal au seuil



### 2.1.5 Processus *affichage*

Le processus *affichage* reçoit sur son canal d'entrée un entier puis l'affiche

## 2.2 Prise en compte d'un tableau de pixels comme source

Plutôt que d'analyser un seul pixel, on souhaite bien sûr opérer sur des images (provenant par exemple d'une camera). Pour cela il faut faire en sorte que tous les processus répètent leur action sur les pixels de l'image.

### 2.2.1 Intégration d'un tableau

1. Modifiez la fonction *source* :
  - (a) Déclarez et initialisez en début de fonction un tableau unidimensionnel d'entiers constant (nommé *image*) de la manière suivante :

```
image := [9]int{100, 200, 150, 32, 250, 18, 47, 242, 99}
```
  - (b) Incluez l'envoi de la valeur d'un pixel dans une boucle qui itère sur les éléments du tableau. Vous pouvez utiliser `for i := 0 ; i < 9 ; i++{ }` pour itérer sur les indices du tableau.
2. Modifiez les fonctions *seuillage* et *affichage* afin qu'elles puissent traiter l'ensemble des pixels de l'image en utilisant une boucle, dont le nombre de tour est égal à la taille du tableau (codée en dur, ce n'est pas générique mais ce n'est pas gênant pour la suite)

### 2.2.2 Boucles d'itération sur tableau

Dans la fonction *source*, mettez en œuvre et testez les 2 variantes de boucles suivantes :

1. `for i := range image { }`, qui itère sur les indices du tableau *image*
2. `for i,e := range image { }`, qui itère sur les indices (*i*) et les éléments (*e*) du tableau *image*. Si *i* n'est pas utilisé, il faut le remplacer par un `_`, ce qui donne : `for _,e := range image { }`

### 2.2.3 Boucle infinie

1. Remplacez la boucle finie dans les fonction *seuillage* et *affichage* par une boucle infinie : `for{ }`
2. Que constatez-vous ? Le résultat est-il bien produit ? peut-on dire que le système fonctionne ?

## 2.3 Adaptation en système fonctionnant sans arrêt

On suppose que notre système doit opérer sur un flot d'image infini. On doit donc modifier son comportement pour qu'il fonctionne sans arrêt.

1. Repartez du programme précédent (*seuillage* et *affichage* ont déjà des boucles infinies). Ajoutez une boucle infinie dans *source* afin que l'image soit envoyée continuellement, pour simuler simplement un flot d'image (même si bien sûr dans la réalité ce seront des images différentes). Testez que le programme fonctionne (il sera interrompu automatiquement par l'environnement d'exécution).
2. Les synchronisations entre les goroutines et le main sont-elle toujours nécessaires ? Enlevez-les et testez le programme, que se passe-t-il ?
3. Quelle est la nature du problème, que faudrait-il ajouter dans le main pour résoudre le problème ? Testez vos différentes idées.

## 2.4 Marqueurs de fin de ligne et de fin de tableau

Le système actuel considère le flot d'image comme un flot de pixel infini, sans que l'on puisse en distinguer la structure (fins de lignes et fins d'images) dont la connaissance peut être nécessaire. Pour améliorer cela, on va modifier le tableau *image* pour inclure des marqueur de fin de ligne et de fin d'image, et modifier les fonctions afin qu'elle en tiennent compte.



1. Ajoutez en dehors du *main* la définition des constantes permettant de dénommer symboliquement les marqueurs. Les marqueurs seront des entiers négatifs, comme tous les pixels ont une valeur positive, il n'y aura pas d'ambiguïté :

```
const finLigne = -1
const finImage = -2
```

2. Modifiez la définition de l'image pour inclure les marqueurs : `image := [12]int{100, 200, 150, finLigne, 32, 250, 18, finLigne, 47, 242, 99, finImage}`
3. Modifiez la fonction seuillage : si elle reçoit une valeur égale à *finLigne* ou *finImage*, elle doit envoyer la valeur dans son canal de sortie sans la modifier
4. Modifiez la fonction affichage :
  - (a) si elle reçoit une valeur égale à *finLigne* elle doit passer à la ligne (utiliser `fmt.Println()`)
  - (b) si elle reçoit une valeur égale à *finImage* elle doit passer 2 lignes (utiliser `fmt.Println()`)
  - (c) sinon, elle doit afficher la valeur sans passer à la ligne (utiliser `fmt.Print(valeur)`)

Pour les différentes alternatives, vous pouvez utiliser des if-else enchainés. Vous pouvez aussi utiliser un switch-case sans valeur de test, dont on donne l'exemple. L'alternative choisie est la première à être vraie :

```
switch {
  case x == 3 :
    instructions1
  case y < b :
    instructions2
  default :
    instructions3
}
```

### 3 Pipeline avec réplication de processus

#### 3.1 Construction d'un pipeline de trois processus (similaire à la partie 3.2)

Un programme de traitement de type pipeline (flot de données) contient typiquement au moins 3 parties (processus) de comportement différent :

- Un processus qui sert à obtenir les données d'entrée (extraction d'une mémoire, réception d'un périphérique ou d'un autre calcul)
- Un ou plusieurs processus qui réalisent le traitement sur les données d'entrée
- Un processus qui dirige les résultats vers la destination souhaitée (affichage à l'écran, stockage en mémoire, émission vers un autre calcul, ...)

On débute cet exercice par un exemple de traitement simple : on assemble dans le *main* trois processus *source*, *traitement*, *affichage* qui ont les comportements suivants :

*source* : envoie séquentiellement 10 entiers (de 1 à 10) dans son canal de sortie

*traitement* : reçoit répétitivement les entiers provenant du processus *source* dans son canal d'entrée, y ajoute un entier fixe *k* fixé en paramètre dans l'appel de la fonction et envoie le résultat dans son canal de sortie.

*affichage* : reçoit répétitivement les entiers produits par le processus *traitement* dans son canal d'entrée et les affiche à l'écran

1. Écrivez le *main* correspondant à ce système, comprenant la déclaration des canaux de communication et l'appel des fonctions avec leurs paramètres
2. Écrivez les trois fonctions définissant le comportement des processus *source*, *traitement* et *affichage*.



## 3.2 Pipeline paramétrique

On souhaite maintenant appliquer plusieurs traitements successifs ayant le comportement du *processus traitement*, avec des paramètres différents, tout en conservant un fonctionnement en pipeline.

Pour ce faire, on propose de remplacer l'appel du *processus traitement* initial par un ensemble de *nProc* *processus traitement* cascades (les uns à la suite des autres) reliés par des canaux.

On rajoute dans le *main* les déclarations suivantes :

- La déclaration d'une constante *nbProc* qui définit le nombre de processus de traitement :

```
const nbProc = 3
```

- La déclaration et l'allocation du tableau des canaux qui relieront les processus *source*, *traitement* et *affichage* :

```
var tabCan [nbProc+1] chan int // déclaration
for i := range tabCan {
    tabCan[i] = make(chan int) }
```

- La déclaration du tableau d'entiers (de taille *nbProc*) qui contiendra les paramètres pour chacun des processus *traitement* (car les paramètres sont différents) :

```
tabK := [3]int{2, 3, 4}
```

1. Faites un schéma du système
2. Modifiez le *main* pour inclure ces déclarations, et lancer *nProc* processus *traitement* (ainsi qu'un processus *source* et un processus *affichage*) en utilisant les variables déclarées.

## 3.3 Pipeline de calcul d'une expression récurrente

### 3.3.1 Présentation

En s'aidant de la structure développée dans l'exercice précédent, on souhaite construire à l'aide d'une réplique parallèle un pipeline qui calcule le polynôme suivant (filtre sigma) :

$$P_n(x) = \sum_{k=1}^n \frac{x^k}{k}$$

Pour cela on utilise la formule de récurrence suivante :

$$\begin{aligned} PU_k &:= PU_{k-1} \times x \\ S_k &:= S_{k-1} + \frac{PU_k}{k} \end{aligned}$$

Avec les initialisations :

$$\begin{aligned} PU_0 &= 1 \\ S_0 &= 0 \end{aligned}$$

*k* désigne l'étape courante de calcul, définie en fonction de l'étape *k - 1*. *PU<sub>k</sub>* désigne la puissance *k* d'un échantillon *x*, calculée par récurrence, et *S<sub>k</sub>* désigne la somme partielle de monômes, également calculée par récurrence. Un processus de traitement est associé à chaque étape de calcul (le premier processus de traitement calcule l'étape pour *k=1*).

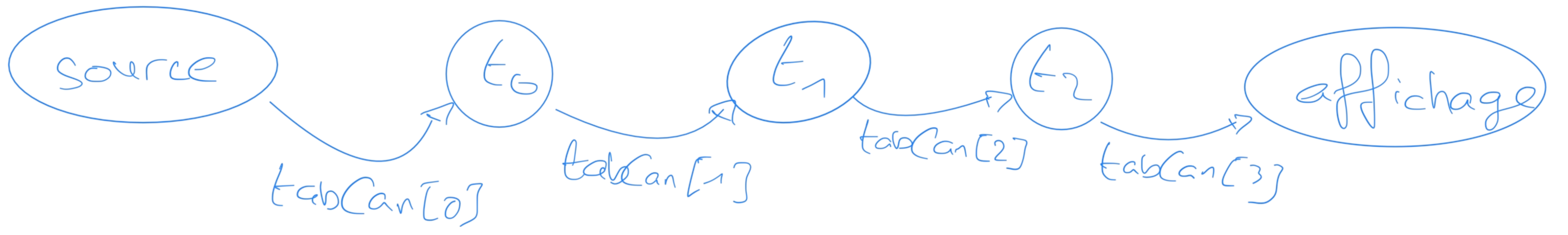
On met en œuvre ce filtre en envoyant **séquentiellement** sur le canal d'entrée d'un processus de traitement 3 valeurs qui sont :

- *x*, un échantillon courant,
- *PU<sub>k</sub>*, la puissance courante de *x* (*x<sup>k</sup>*),
- *S<sub>k</sub>*, la somme courante.



3.2.

$t = \text{traitemnt}$





### 3.3.2 Mise en œuvre

En vous inspirant des exercices précédents, implémentez un programme réalisant ce calcul :

- Ajoutez des déclarations de constantes (en début de programme) :
  - l'ordre du polynôme ( $n$ ) : par exemple `const ordre = 3`
  - le nombre d'échantillons  $nbX$  (qui est le nombre de résultats à calculer) : par exemple `const nbX = 10`
- Modifiez le typage des canaux et des variables concernées : il faut utiliser `float64`, vu la nature du calcul. Pour les instructions où vous avez besoin de convertir un entier en flottant (dans *source* pour l'indice de boucle, dans *traitement* pour la valeur de  $k$ ) utilisez la conversion explicite : `float64(i)`
- Modifiez le processus *source*, qui produit les valeurs initiales du pipeline pour chaque échantillon : pour la valeur  $x$ , on envoie par exemple l'indice d'une boucle séquentielle (comme pour le processus *source* initial); pour les 2 autres valeurs, on envoie les valeurs initiales du calcul, telles que définies dans la partie 3.1.
- Modifiez le processus *affichage* : pour chaque échantillon il reçoit 3 valeurs et doit afficher le résultat.
- Modifiez le processus *traitement*, qui, à chaque tour de boucle, reçoit 3 valeurs ( $x$ , PU,  $S$ ) séquentiellement via le même canal, effectue le calcul, et envoie 3 valeurs au processus suivant. On conserve le paramètre  $k$ , pour un usage différent des parties 1 et 2 : comme indiqué dans la partie 3.1,  $k$  désigne l'étape du calcul associée au processus, et est utilisé pour le calcul de  $S_k$
- Modifiez le programme principal, notamment en fixant la valeur du paramètre  $k$  pour chaque processus *traitement* (sans utiliser un tableau comme dans la partie 2).

### 3.3.3 Question bonus : vérification des calculs

Il est bien sûr difficile de vérifier si la trace est correcte sans faire le calcul associé. Afin de vérifier que votre système est correct, vous devez comparer les résultats avec ceux provenant d'une autre méthode :

1. Écrivez une fonction qui calcule et retourne la valeur de  $P$  pour des valeurs de  $x$  et  $n$  données en paramètre
2. Intégrez un appel à cette fonction dans la fonction *affichage*, de façon à ce que celle-ci compare les 2 résultats pour toutes les valeurs et alerte en cas de différence constatée.