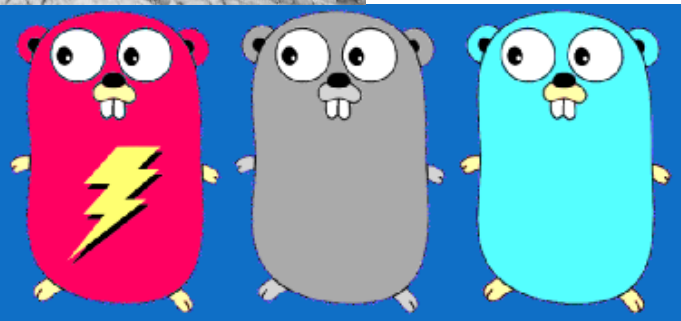
UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY



#3

09/12/2025

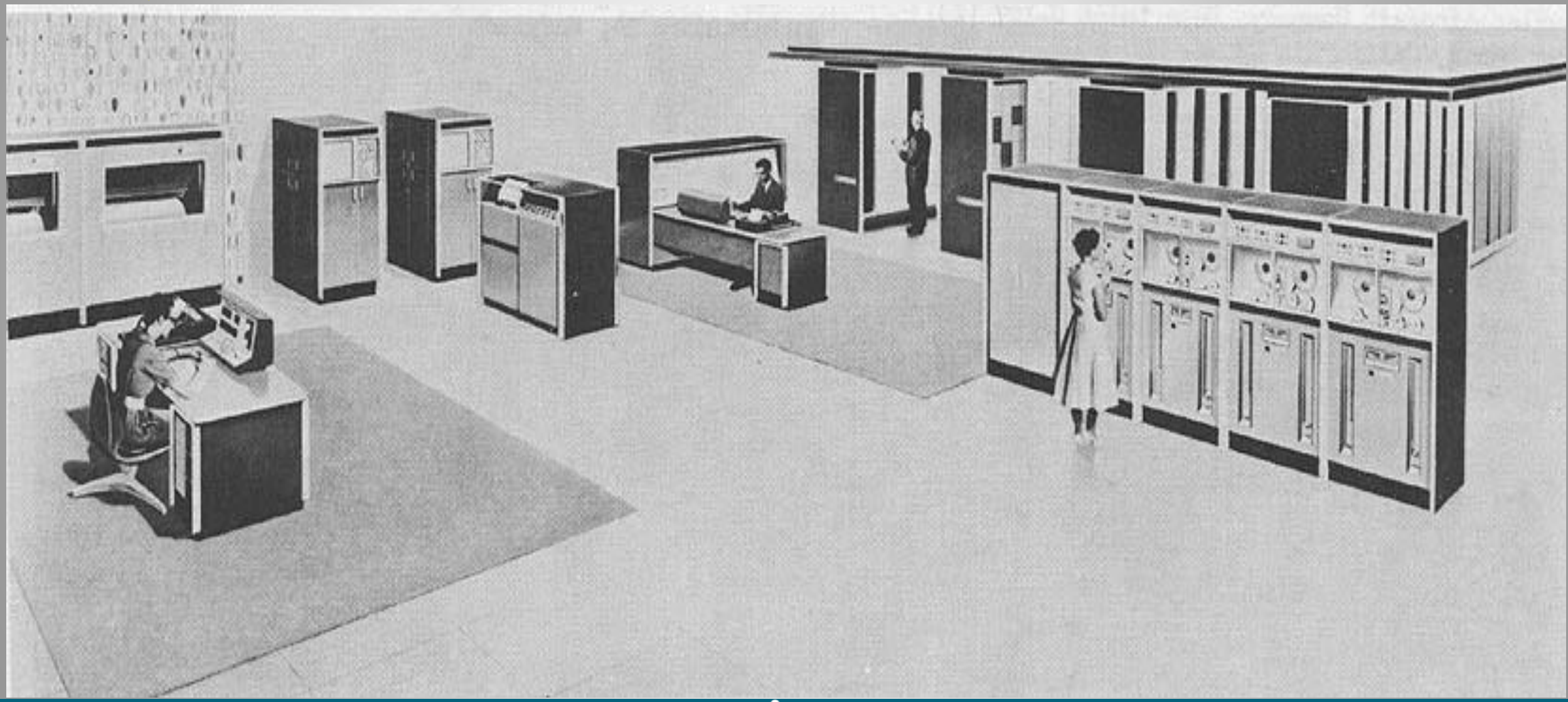
jean-michel.batto@cea.fr

cea

https://gogs.eldarsoft.com/M2_IHPS

Présentation du travail par campagne / SLURM

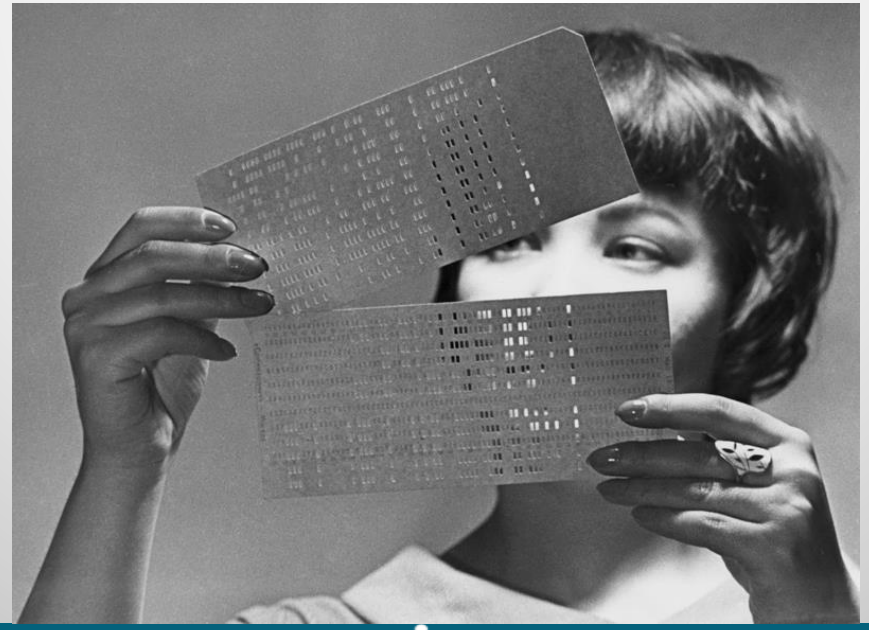
- https://slurm.schedmd.com



50's

- Premières générations d'ordinateurs
  - La programmation est faite à base de cartes perforées, regroupées par lot (batch).

    Les cartes les plus répandues ont 80 colonnes et 12 « lignes ».
  - Les entrées sont elles aussi fournies au travers de cartes, puis de bandes magnétiques.
  - Les sorties sont réalisées sur cartes ou imprimantes.



60 ’s

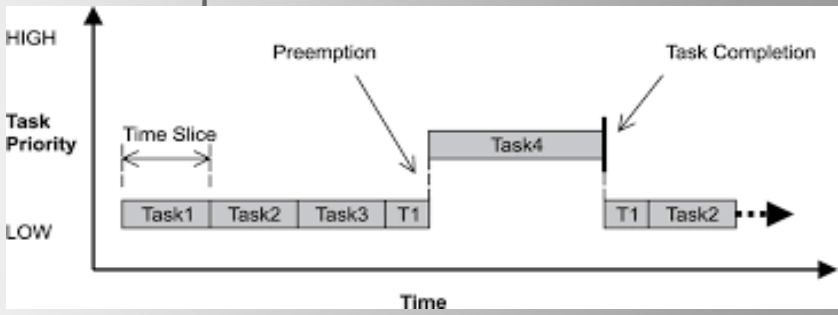
- Premières générations d’ordinateurs
  - L’utilisation des machines est assurée par des opérateurs qui chargent les paquets de cartes en machine suivant un planning pré-établi.
    - → batch scheduling
  - Les résultats, des « listings » papiers, sont fournis aux utilisateurs après l’exécution de leurs « jobs ».
    - Les « jobs » en erreur produisent une quantité pharamineuse de sorties

60's

- Premières générations d'ordinateurs
  - Mode d'utilisation
    - On planifie (schedule) l'exécution de jobs par paquets consécutifs de cartes perforées.
    - On génère des « listings » papiers.
  - On passe un temps certain à mettre au point les « cartes » de ses « jobs » et à en traiter les « listings »
- L'émergence de l'informatique moderne
  - L'arrivée des transistors, des bandes magnétiques et des mémoires permet la conception de nouvelles machines.
  - Les « mainframes » apparaissent
    - Les terminaux « graphiques » 80 colonnes font leur entrée.
  - Des lecteurs de cartes restent associés…
    - Pour réutiliser les codes…Et migrer vers des « scripts »

L'ère « mainframe » (70's)

- Les scripts et programmes se numérisent
  - Les cartes sont mises au placard après numérisation.
  - Les données sont enregistrées sur bandes magnétiques et chargées/écrites depuis les programmes.
- Les « jobs » sont planifiés par des opérateurs puis par des applications spécialisées.
  - Les premiers « batch scheduler »…
- Mode d'utilisation
  - « Soumission » de scripts batch par les utilisateurs (jobs).
  - Ordonnancement automatique de l'exécution des jobs par une application dédiée.
  - On génère des « listings » numérisés : sorties « écran » redirigées dans des fichiers.
- On gagne du temps dans la mise au point des scripts et programmes et le dépouillement des résultats.
- On attend en fonction de l'importance du programme plus ou moins longtemps avant son exécution.

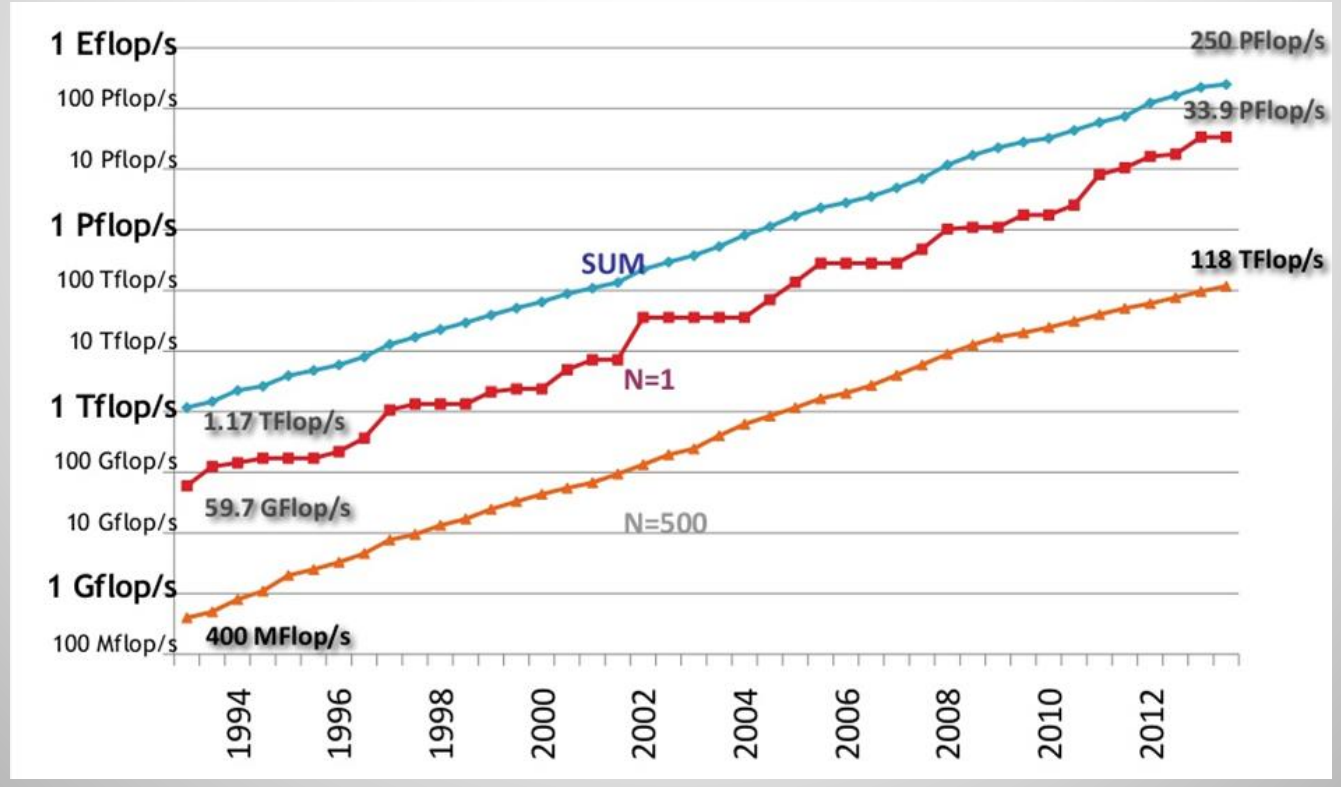


L'ère « PC » (80's - 2000's)

- L'informatique se miniaturise et se démocratise par le canal des « personnal computeur »
- Qui reprennent les concepts des « mainframe » en associant directement le terminal à l' « unité centrale ».
- L'évolution est forte et rapide.
  - Les interfaces graphiques
  - font vite leur entrée
- Les systèmes d'exploitation permettent (Unix 1971, Linux 1991)
  - Une interaction directe via des interfaces graphiques simplifiant l'utilisation des machines
  - Une interaction en mode ligne de commandes et/ou scripts.
    - Ex : fichiers script « .bat » de Windows
  - Les « scripts » restent exécutables en arrière plan (crontab) pour les traitements « batch ».

L'émergence des clusters (90's)

- Les réseaux prennent de l'ampleur et permettent une interconnexion performante d'unités individuelles type PC.
- Le HPC s'engouffre dans cette voie face à la diminution des performances des approches monolithiques des « Mainframe ».
  - Le nombre d'unités de calcul connectées ne cessera de croître...



L'émergence des clusters (90's)

- L'utilisation des clusters nécessite alors l'orchestration de plusieurs unités de calcul indépendantes.
  - Notion de « **jobs parallèles** » exécutés sur des « systèmes distribués »
- Un ordonnanceur central est en charge de la répartition « spatiale » et « temporelle » des travaux.
  - Dédier un certain nombre de « nœuds » pour une période de temps donnée à un « job ».
- Les jobs deviennent hétérogènes
  - Une ou plusieurs sections parallèles permettant l'exécution de codes de calcul optimisés pour l'utilisation de plusieurs unités de calcul
    - Émergence du modèle MPI ! (CSP)
  - Encapsulée(s) dans le déroulement classique du script « batch »
- L'ordonnancement se complexifie
  - Différents besoins en terme de nombres d'unités de calcul dans les sections parallèles.
  - Différentes localités.
- Les « batch scheduler » évoluent donc pour traiter efficacement ces « systèmes distribués »
  - On parle maintenant de **DRMS (Distributed Resource Management System) – PBS, Torque, SGE**
  - **Concept de pilotage (Pilot-Job)**

Principe d'un DRMS

- Rappels
  - Evolution des batchs scheduler « initiaux »
    - Gérant principalement des « jobs » en **time slicing**

      → **composant « job manager »**
    - Prise en charge d'une quantité de ressources de calcul grandissante et distribuée

      → **composant « resource manager »**

Usage d'un DRMS

- Repose généralement sur un composant **leader** central
  - Permettant aux utilisateurs d'enregistrer leurs « jobs » pour exécution ultérieure
  - Fournissant un statut des ressources disponibles et en cours d'utilisation
  - Fournissant un statut des jobs en cours de calcul ou en attente de ressources
  - Fournissant l'historique et les statistiques d'utilisation des ressources

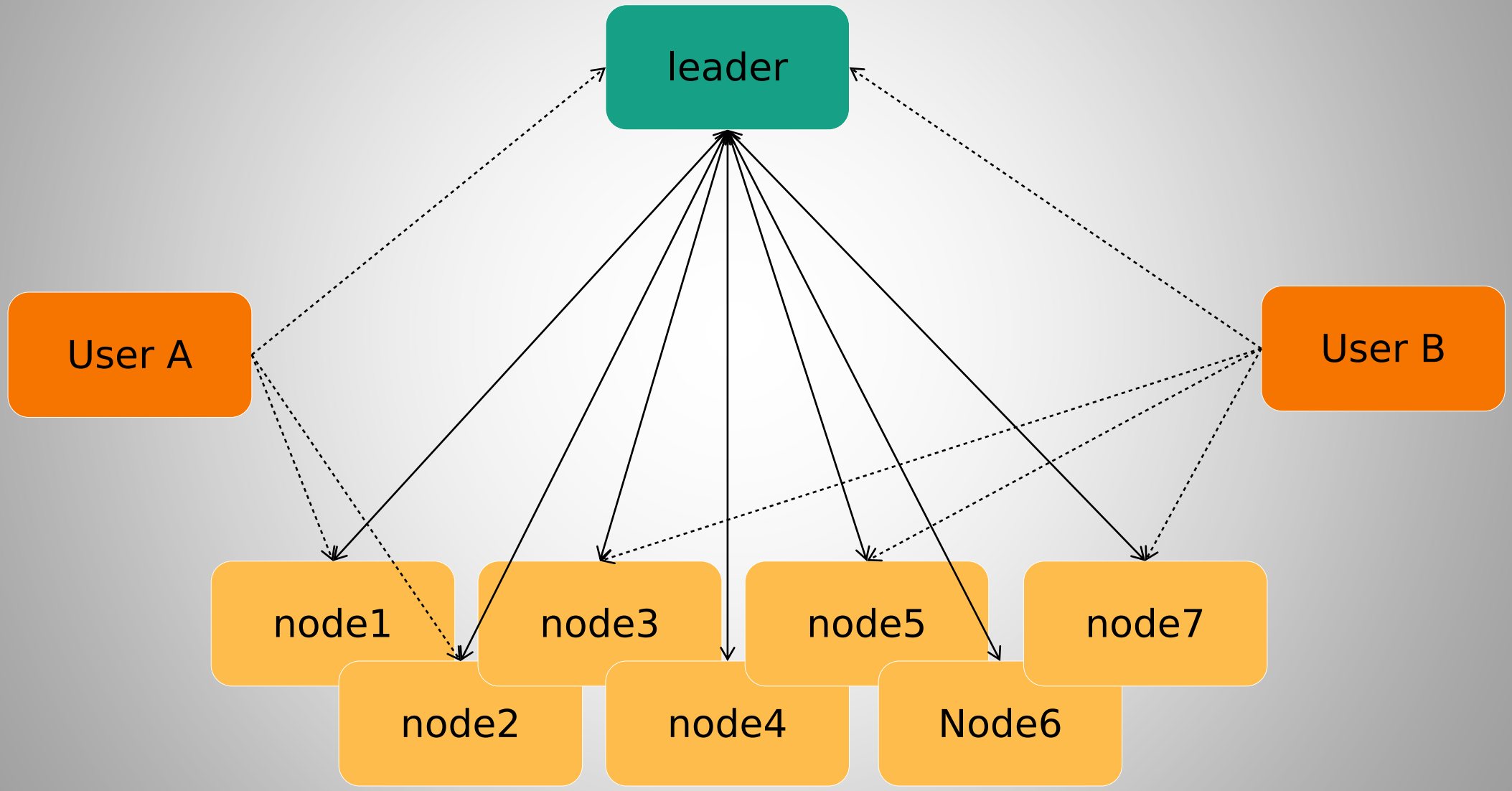
Architecture d'un DRMS

- Repose généralement sur un composant **leader** central
  - Orchestrant la répartition des ressources entre les « jobs » au cours du temps
  - Orchestrant la mise en exécution, l'arrêt des jobs ainsi que le suivi de la bonne utilisation et la libération des ressources utilisées

Architecture d'un DRMS

- Repose généralement sur un ensemble de **workers** distribués
  - Généralement un par nœud de calcul
  - Fournit l'état du nœud au leader et permet les intéractions directes avec celui-ci ou les utilisateurs
  - En charge du démarrage des exécutions de scripts et ou d'applications pour les utilisateurs
  - Assure le suivi de la bonne utilisation et la libération des ressources utilisées

Architecture DRMS (Pilot-Job)



Un DRMS : SLURM

- Simple Linux Utility for Resource Management
  - Simple → Scalable
- Projet démarré au LLNL en 2002
  - Lawrence Livermore National Laboratory, Livermore, CA, USA
- Continué par SchedMD depuis 2010
  - Entreprise créé par les deux développeurs principaux
- Produit OpenSource écrit en C : Licence GPLv2
- Utilisable sur la majorité des environnements de type UNIX
  - AIX, Linux, BSD, ...
- Utilisé sur une multitude de grands calculateurs à travers le monde
  - parmi les plus grands

slurm workload manager

Objectifs

- Scalable
  - Permet la gestion de plusieurs dizaines de milliers de nœuds
  - Permet la gestion de plusieurs centaines de milliers de cœurs de calcul
- Modulaire
  - Basé sur la notion de plugins pour spécialiser différentes parties du produit en fonction des besoins

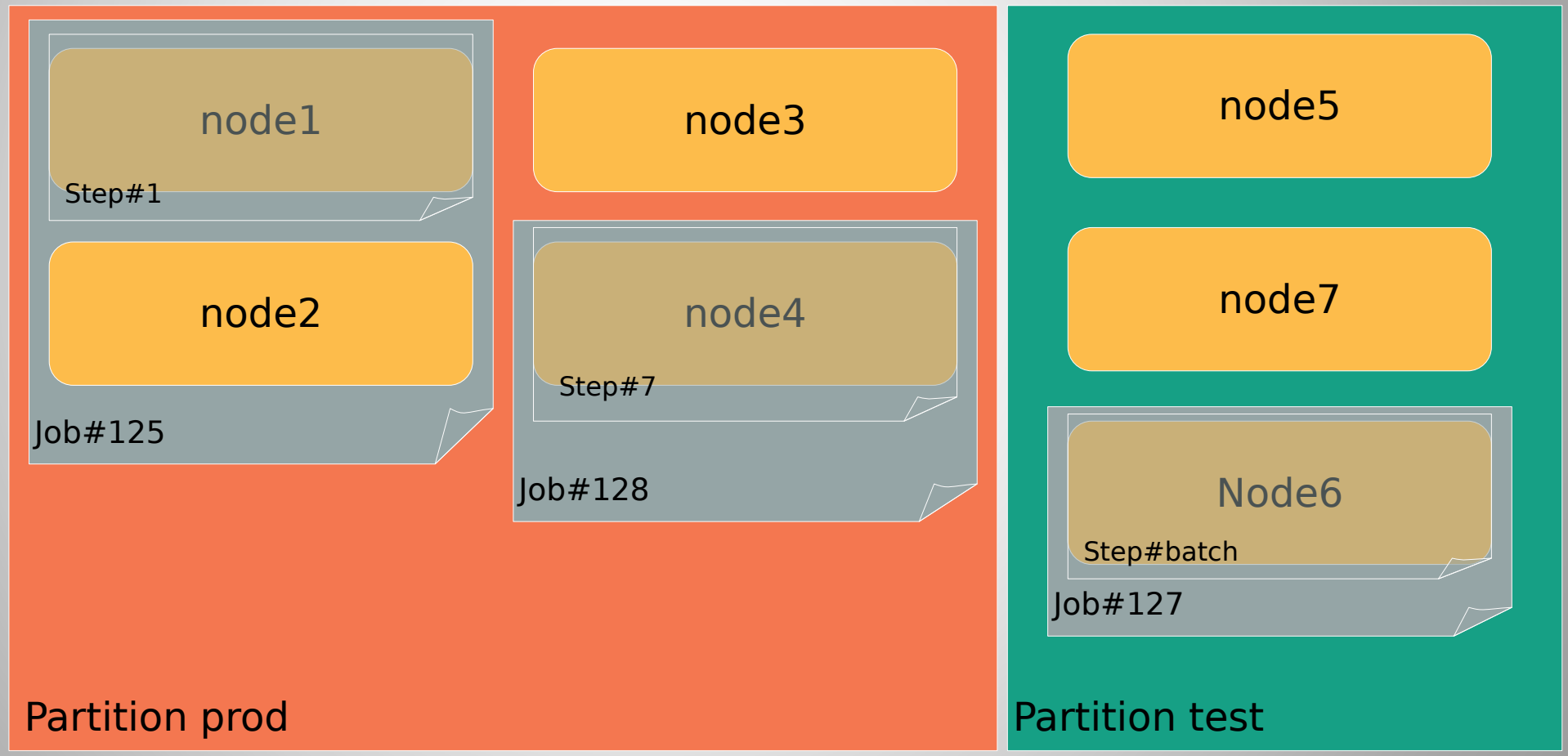
Composants

- slurmctld
  - Composant « leader » (controler)
- slurmdbd
  - Composant additionnel au «leader » pour la persistence des données de comptabilité sur les jobs et la gestion des utilisateurs et de leurs droits
    - Backend MariaDB/Mysql nécessaire
- slurmd
  - Composant « worker »

Concepts

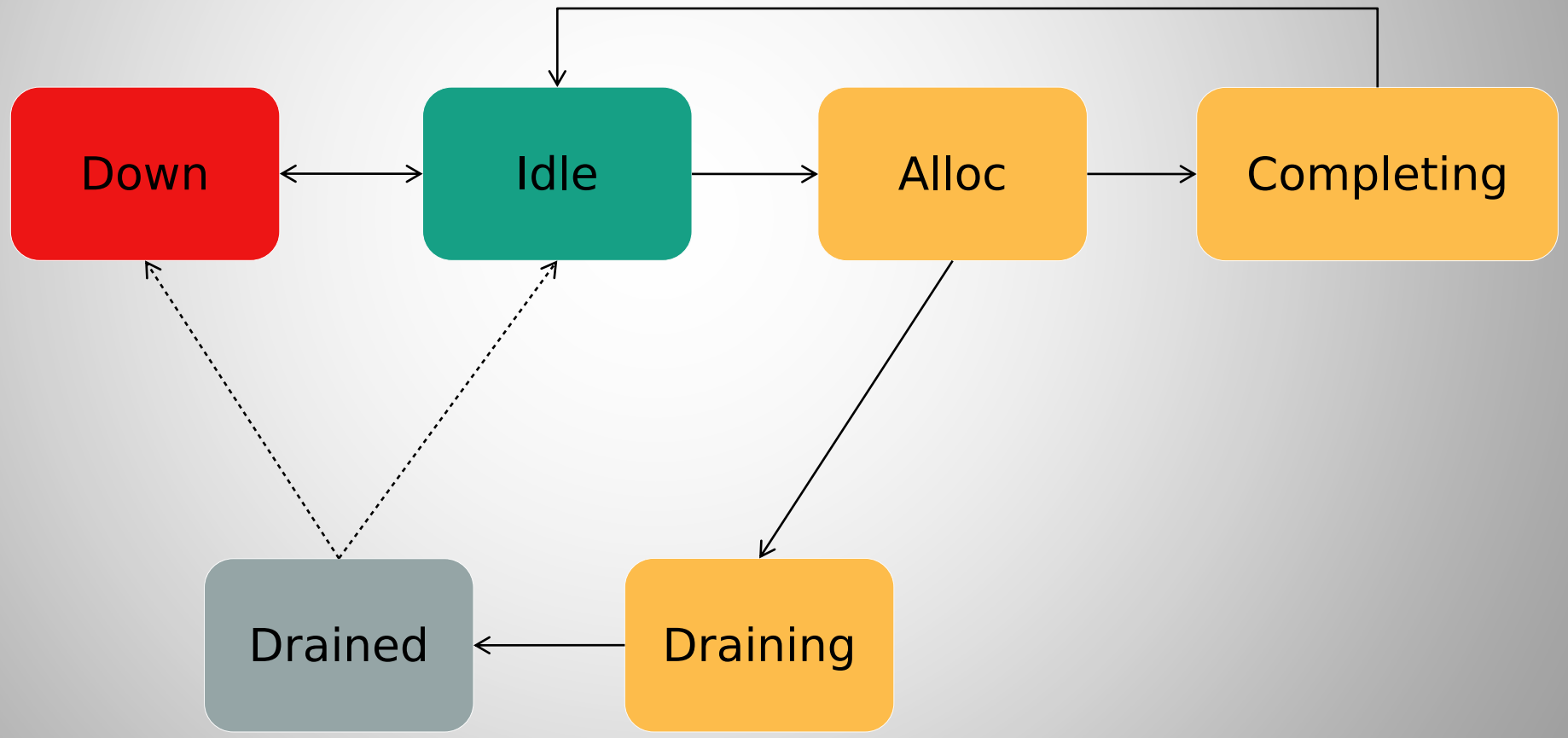
- Partition
  - « Pool » de nœuds utilisables au sein d'un même « job »
    - Un nœud peut appartenir à plusieurs partitions
- Node
  - Unité indépendante fournissant des ressources utilisables par les utilisateurs
    - Sockets/Cores/Threads, Memory, GPUs, …
- Job
  - Demande d'allocation de ressources *dans une partition* associée à un utilisateur
    - Ensemble de ressources réparties sur des nœuds pour un temps défini
    - Batch (script fourni) ou Interactif (shell)
- Jobstep
  - Demande de sous-allocation de ressources pour effectuer une tâche particulière
    - Sous-ensemble de ressources parmi les ressources allouées pour le job associé

Articulation



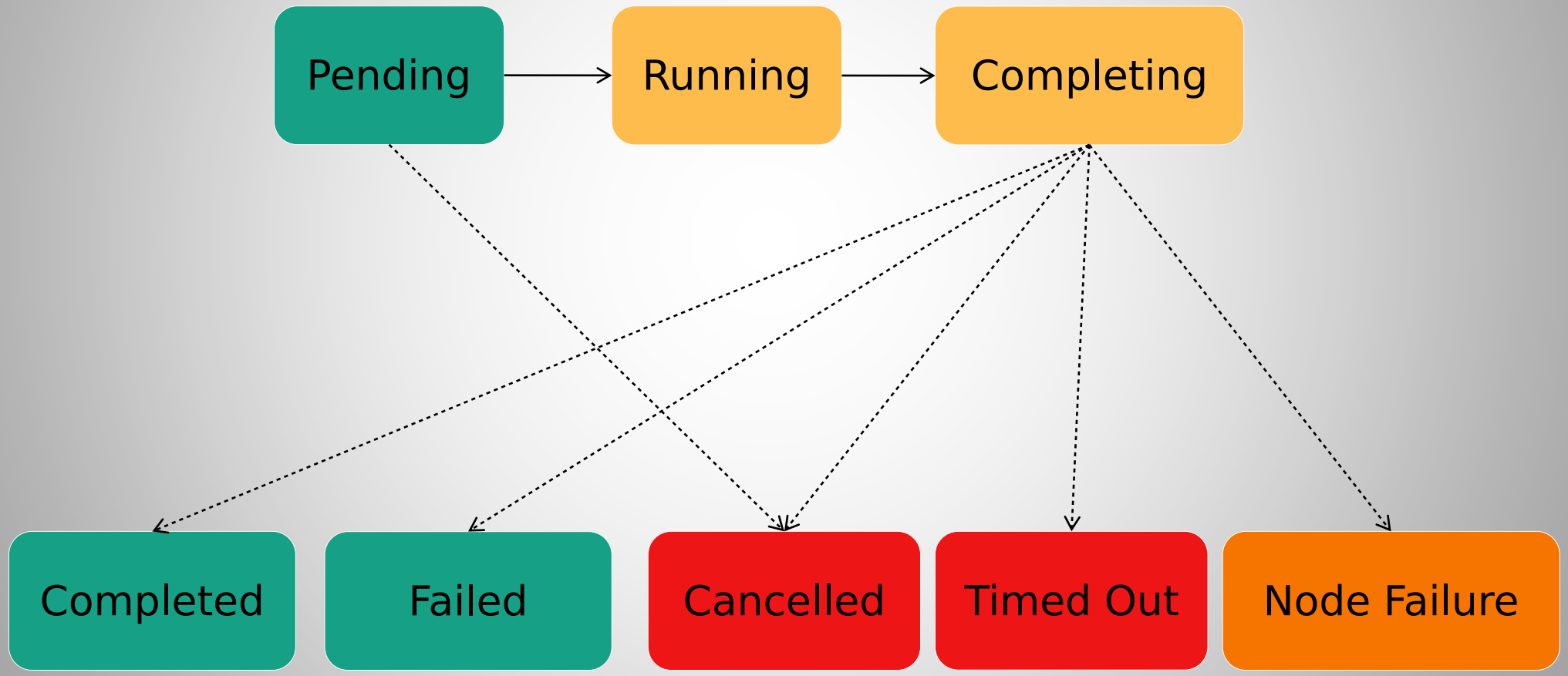
Node

Node « states »

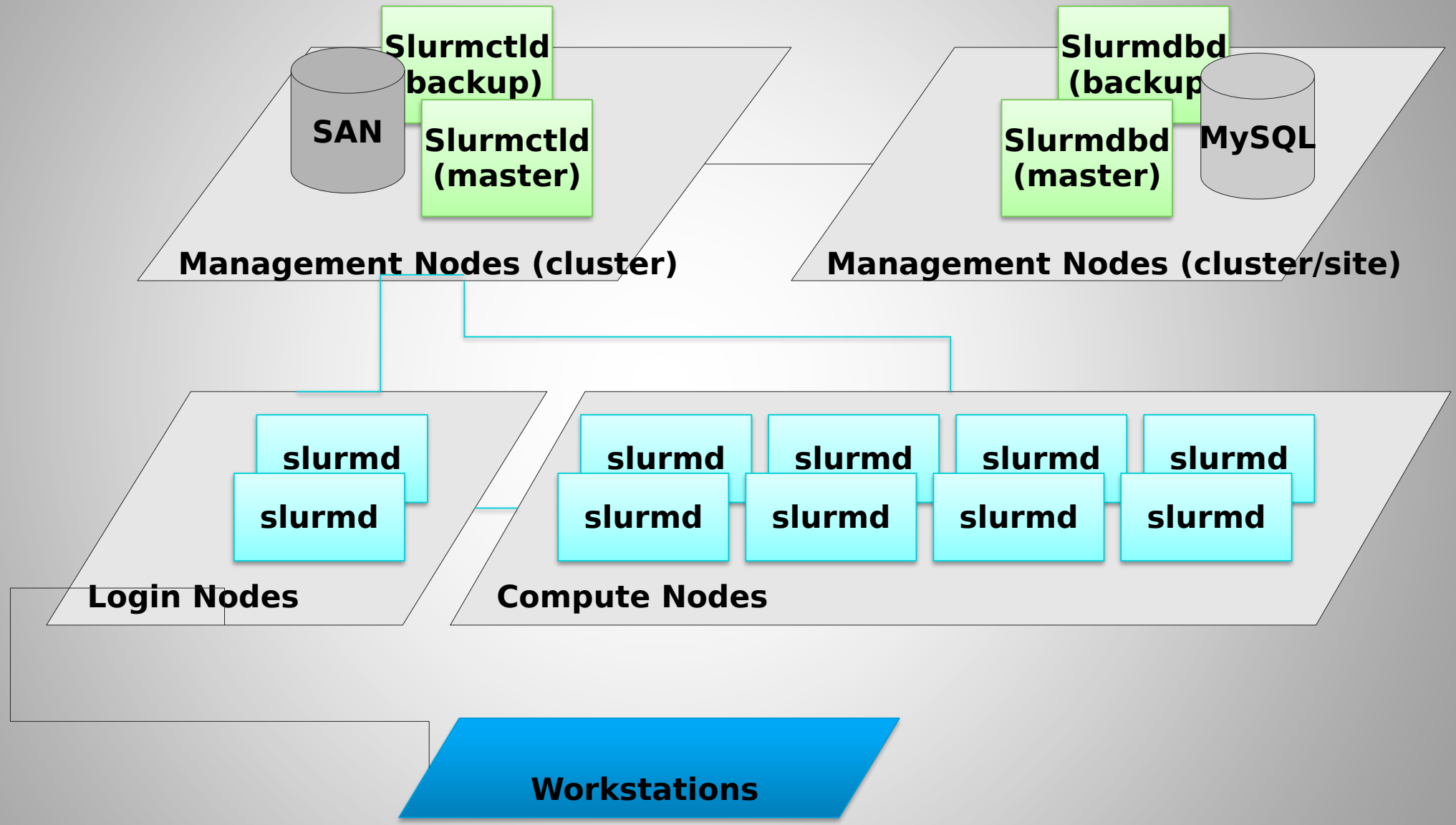


Job

Job « states »



Organisation physique

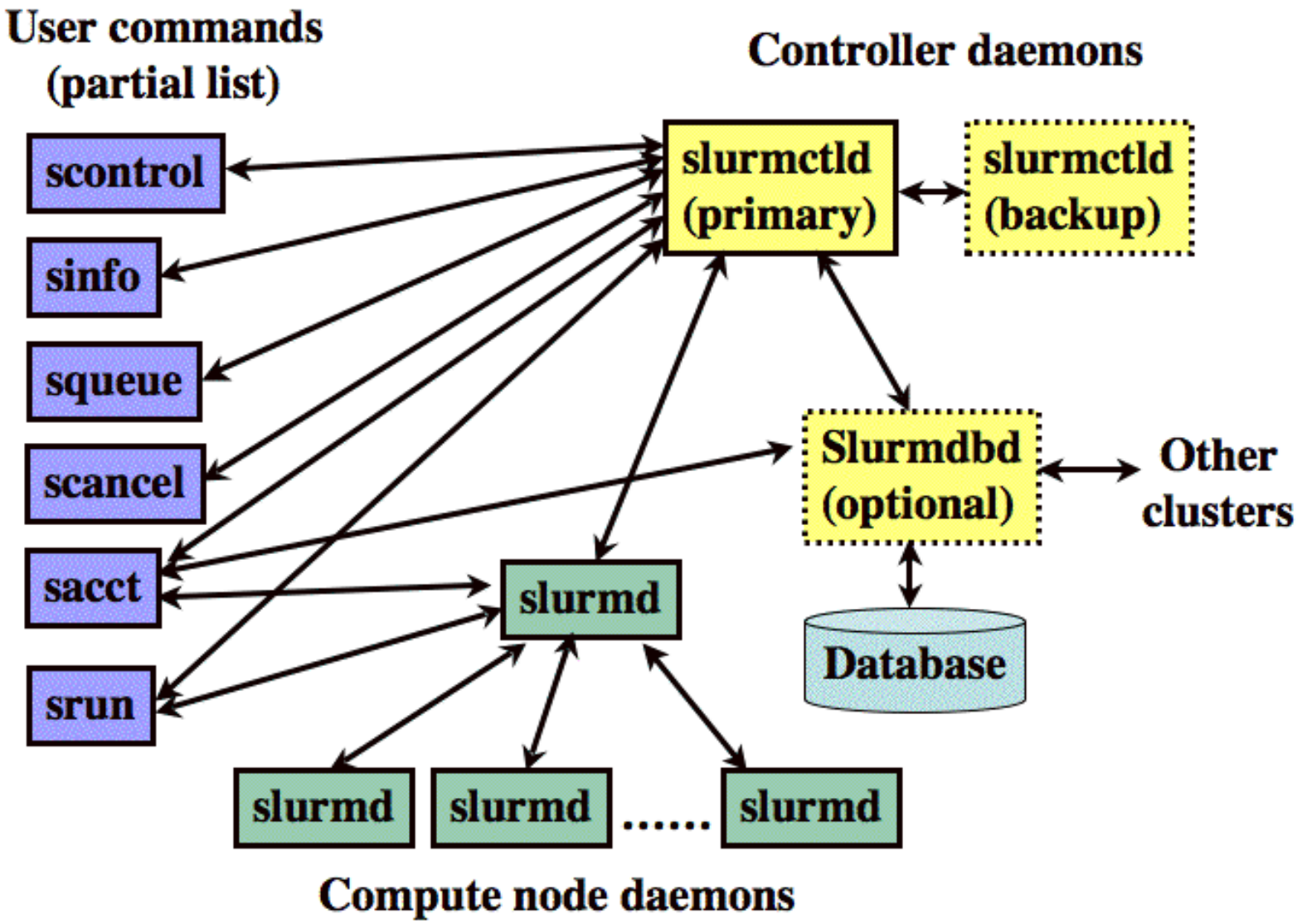


Commandes principales

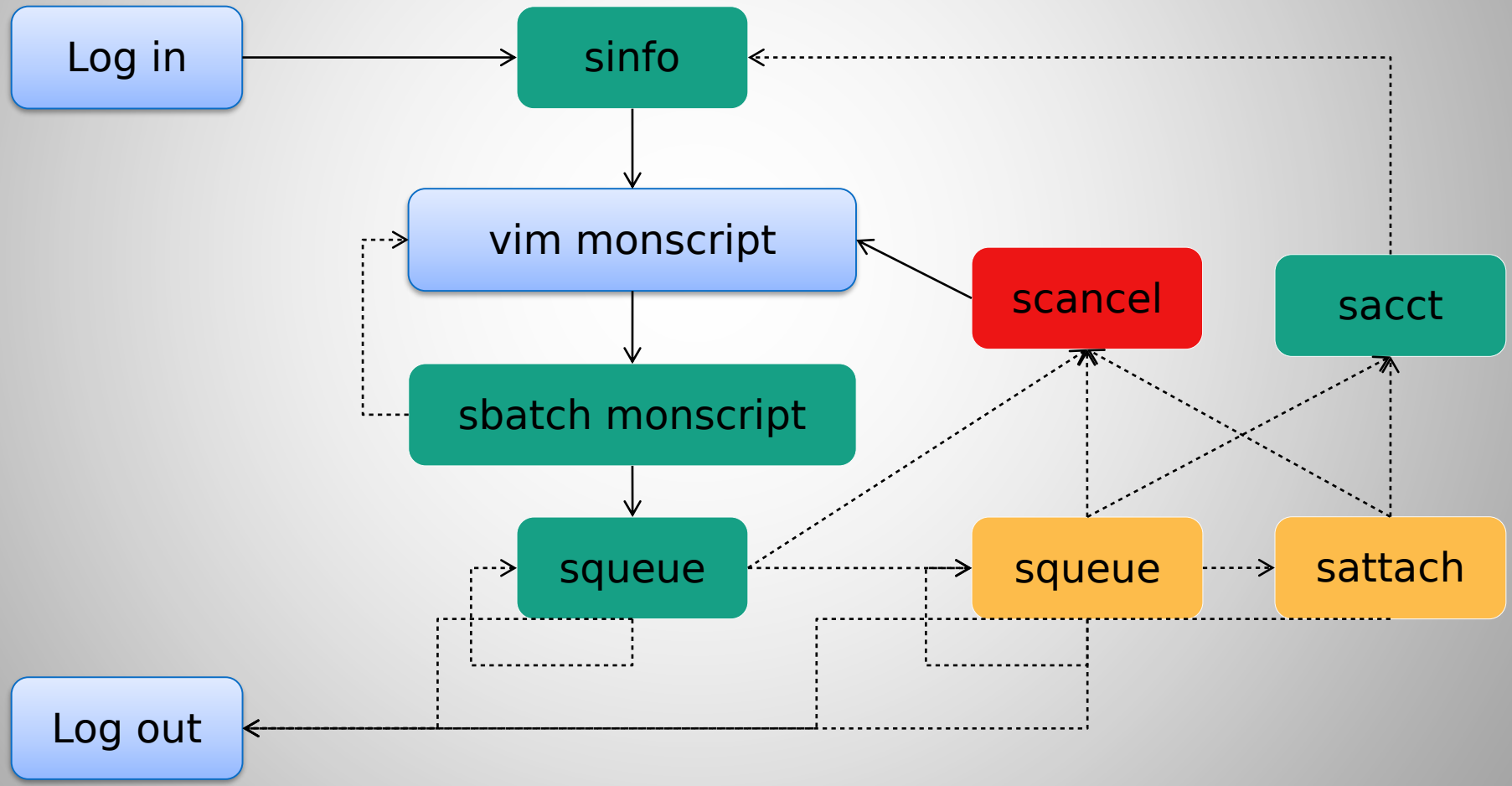
- **scontrol**
  - obtention & modification de la configuration
  - obtention & modification des états des éléments (nodes, partitions, jobs, …)
- **sacctmgr**
  - obtention & modification de la configuration des éléments stockés en BD (« users », « accounts », « qos » …)
- **sinfo**
  - Information sur l'état des partitions
- **squeue**
  - Information sur l'état des « jobs »
- **sacct**
  - Information sur l'exécution de jobs en cours ou passés
- **sstat**
  - Information détaillée sur l'exécution de jobs en cours

- **sbatch**
  - « Soumission » d'une demande d'allocation de ressources *détaillant les ressources nécessaires*
  - Fourniture du « script batch » associé
    - Exécution du script sur les ressources disponibles sur le premier nœud « alloué »
  - Mode « batch » (non interactif)
    - L'utilisateur ne peut plus interagir directement avec son job et doit utiliser les commandes Slurm adhoc pour cela
    - Les sorties stdout/stderr du script exécuté sont redirigés vers des fichiers (configurables)
- **salloc**
  - « Soumission » d'une demande d'allocation de ressources *détaillant les ressources nécessaires*
  - Lancement d'un shell interactif associé aux ressources allouées dès réalisation ou exécution locale d'un script passé en argument
  - Permet l'exécution de commandes « srun » ultérieures pour créer des « jobstep » dans le job réalisé
    - Facilite les tests en évitant l'attente « pending→running » inhérente à chaque soumission
- **srun**
  - « Soumission » d'une demande d'allocation de ressources *détaillant les ressources nécessaires*
  - Exécution d'un certain nombre de processus répartis sur les ressources allouées
    - en fonction des détails fournis en argument
  - Mode d'utilisation interactif (-s)
    - L'utilisateur suit l'exécution du job dans son terminal et peut interagir avec lui (signaux, stdin, ...)

- **sattach**
  - Permet de suivre et/ou d'interagir avec un job batch à la manière d'un job interactif
- **scancel**
  - Permet la transmission d'un signal à un job ou jobstep
  - Permet de demander la terminaison au plus tôt d'un job ou jobstep

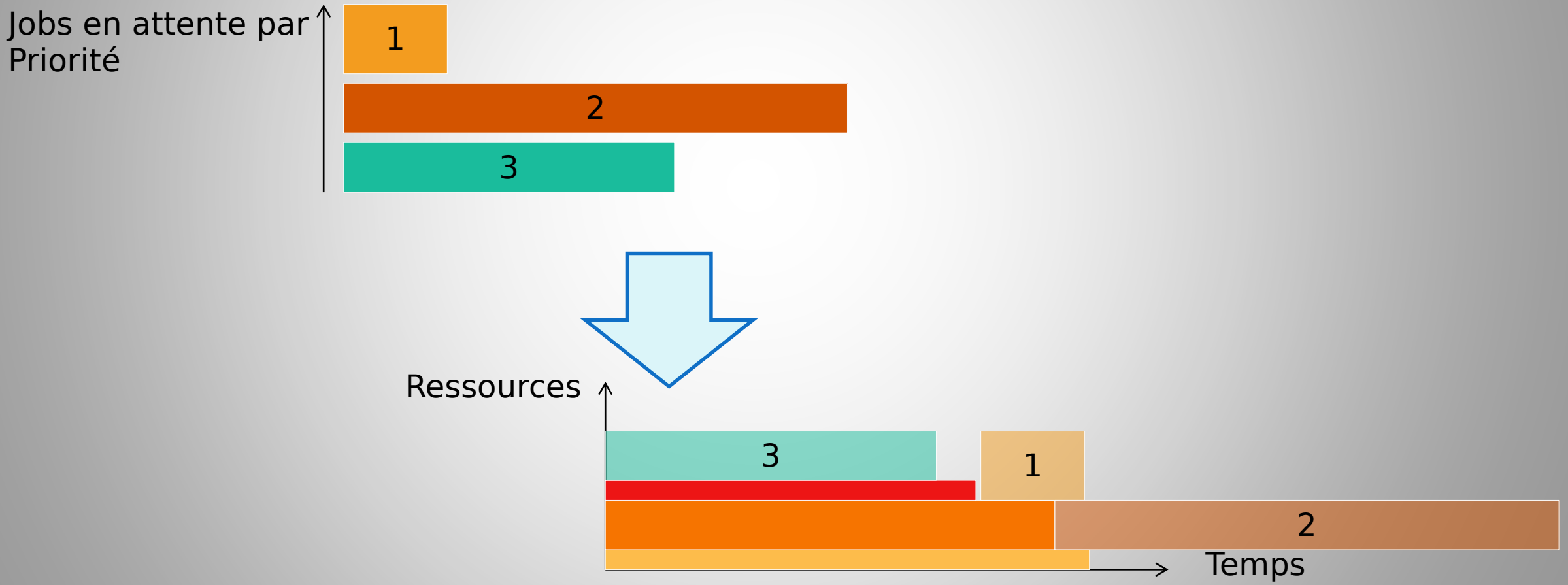

User commands
(partial list)
Controller daemons
scontrol
sinfo
squeue
scancel
sacct
srun
slurmctld
(primary)
slurmctld
(backup)
Slurmdbd
(optional)
Other
clusters
Database
slurmd
slurmd
slurmd
.....
slurmd
Compute node daemons

sbatch


Log in
sinfo
vim monscript
scancel
sacct
sbatch monscript
squeue
squeue
sattach
Log out

Ordonnancement

Comment organiser les priorités ?



Politiques d'ordonnancement classiques

- FIFO : First-In First-Out
  - Premier arrivé, premier servi
- ~~First-Fit~~
  - ~~Le premier qui tient dans l'espace disponible rentre en machine~~
- FairSharing
  - Basé sur une notion de parts de ressources attribués aux différents utilisateurs
  - Celui qui rentre est celui dont l'utilisation est la plus inférieure à ce qu'il est autorisé à utiliser

Politiques d'ordonnancement classiques

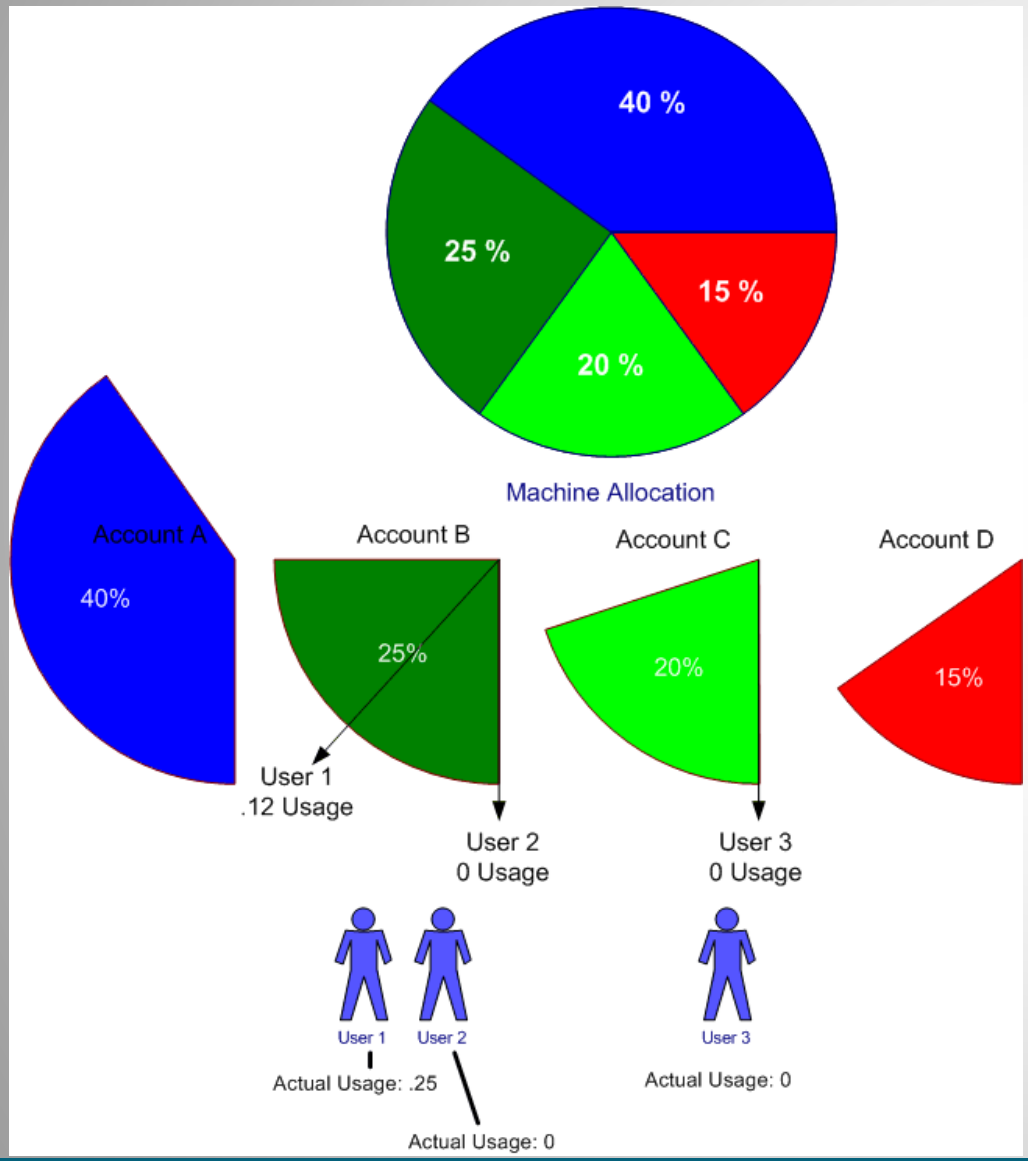
- Aging
  - Le plus ancien est le plus prioritaire
- Size based
  - Le plus petit (ou le plus gros) d'abord
- QOS (Qualité de service)
  - Différentes qualités de service avec différentes restrictions
  - Certaines plus prioritaires que d'autres.
- Backfilling
  - Un job moins prioritaire est exécuté en premier si il ne repousse pas la date de démarrage des plus prioritaires.

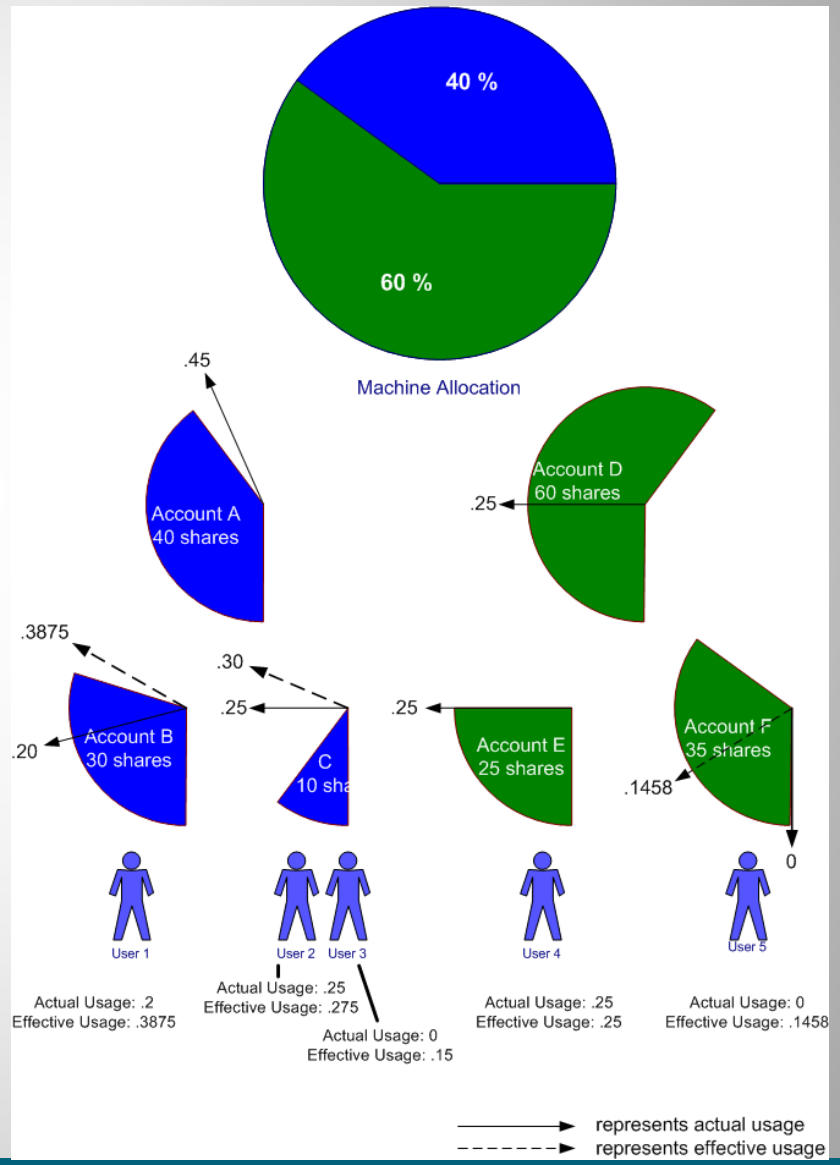
Politiques d'optimisation classique

- Preemption
  - Un job moins prioritaire laisse sa place à un plus prioritaire lorsqu'il doit s'exécuter
    - (suspension d'exécution ou remise en file d'attente (queue))
- Best-effort
  - Un job moins prioritaire est annulé si un plus prioritaire à besoin des ressources

Fairshare

– Groupements hiérarchiques d'utilisateurs, notion d' « account »





FairShare factor

- L’écart entre part utilisable et utilisée des utilisateurs pondère la valeur de chaque job permettant de revenir à l’équilibre souhaité au plus vite
- Ex
  - User-A share=0.3 usage=0.2, fact=0.6
  - User-B share=0.2 usage=0.25, fact=0.45

Partition / QOS Factors

- Chaque partition/qos dispose d'une priorité
- La valeur renvoyée correspond à la normalisation de la valeur de la partition ciblée par rapport à la priorité maximum observée
  - Ex :
    - partition-A priority=20, fact=0.2
    - partition-B priority=100, fact=1.0
    - Partition-C priority=70, fact=0.7

Multifactor priority plugin

- Exemple de configuration

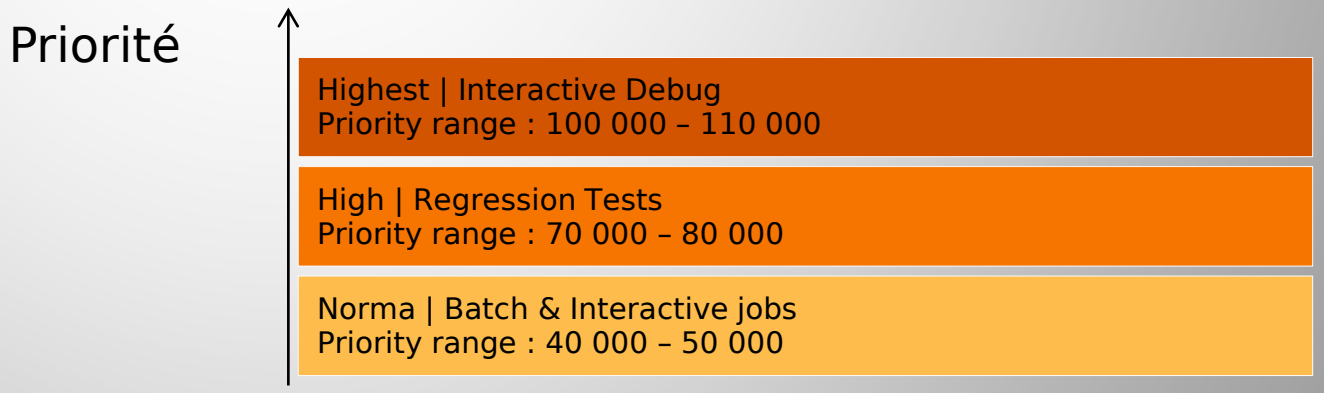
  PriorityWeightQOS=100 000

  PriorityWeightAge=10 000

  PriorityWeightFairshare=10 000

  PriorityWeightJobSize=0

  PriorityWeightPartition=0



Limitation d'accès aux ressources

- Il peut être nécessaire de restreindre l'accès à certaines ressources à certains utilisateurs
- Il peut être nécessaire de restreindre la quantité disponible de ressources pour certains utilisateurs
- Il peut être nécessaire de restreindre le temps d'utilisation maximum possible en fonction des utilisateurs
- Les partitions disposent d'un certain nombre de possibilités de restriction qui ne s'avèrent pas toujours pratiques ou manquent de factorisation
- Les QOS permettent de corriger ce problème en fournissant des restrictions s'appliquant orthogonalement aux partitions
  - Une même partition peut être accédées via différentes QOS
- Les « associations » permettent de raffiner encore la granularité de configuration des limitations
  - Une association peut correspondre à :
    - Un cluster et un account
    - Un cluster, un account et un utilisateur
    - Un cluster, un account, un utilisateur et une partition
  - Les restrictions s'appliquent hiérarchiquement sur les associations d'un utilisateur pour un account donné
    - Un utilisateur peut être associé à plusieurs account

QOS – exemple de limites

- MaxJobsPerUser
  - Quantité maximale de jobs en exécution
- MaxSubmitJobsPerUser
  - Quantité maximale de jobs enregistrés
- MaxNodes
  - Quantité maximale de nœuds utilisables dans un job
- MaxWall
  - Temps d'exécution maximum d'un job
- MaxJobs
  - Quantité maximale de jobs en exécution
- MaxSubmitJobs
  - Quantité maximale de jobs enregistrés
- GrpJobs
  - Quantité maximale de jobs en exécution incluant les jobs de toutes les associations filles
- GrpSubmitJobs : effectif max des jobs en attente

Prise en main de SLURM / TD d'exploration

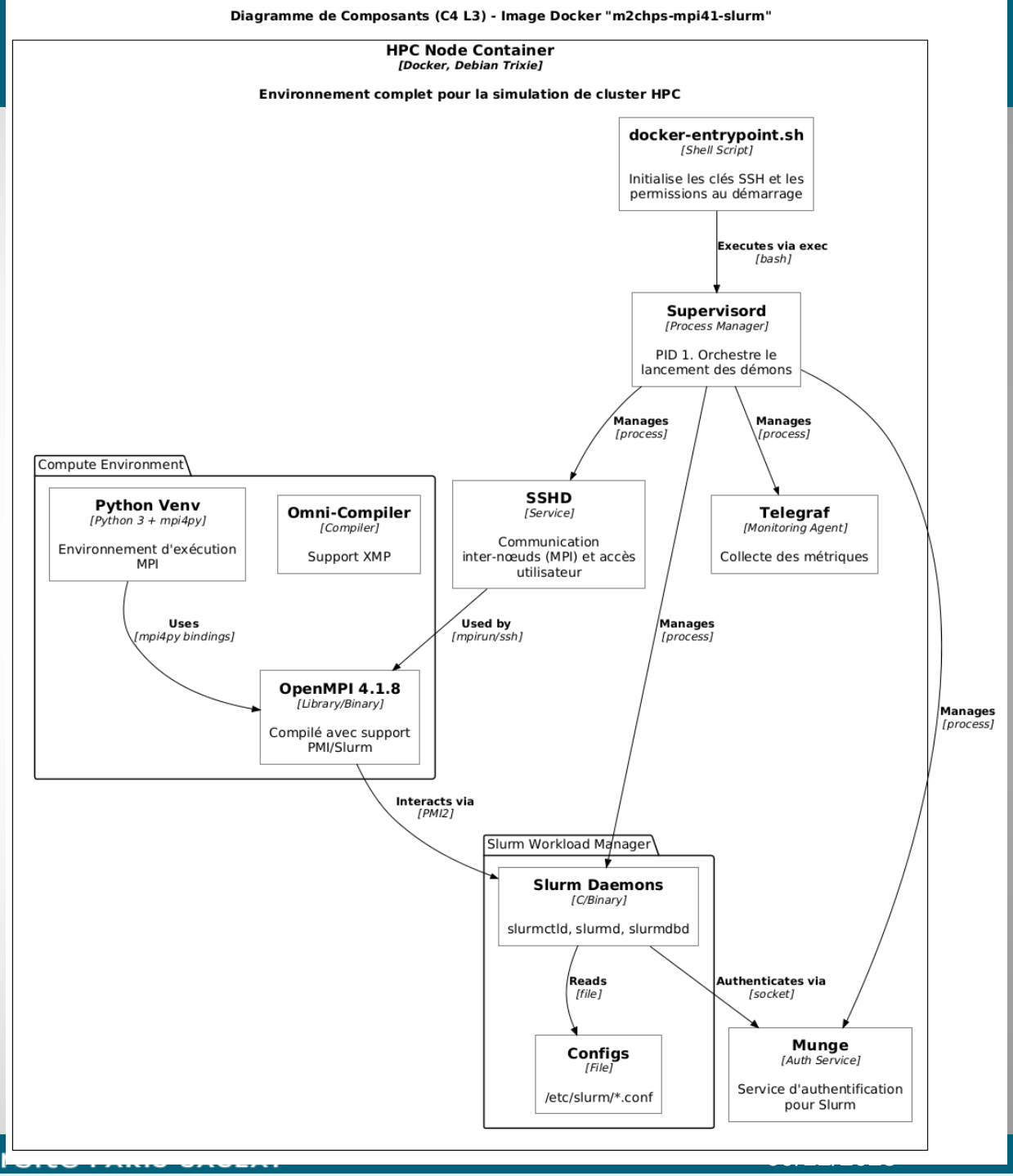
INTERACTIF

- Swarm init ➔ déjà fait dans les TD précédents
- ➔Docker créé un contexte préfixé par le nom du répertoire…
- avec le docker-compose.yml, installation de
  - 1 db mysql pour le nœud slurmdbd,
  - 1 noeud slurmdbd,
  - 1 noeud de contrôle,
  - 2 nœuds de calcul, le tout sous MPI
- **`docker compose up -d`**
- Ouvrir 2 shells (un pour le nœud de contrôle, un pour le nœud C1)

Ce que contient l'image jmbatto/m2chps-mpi41-slurm

Slurm (3 instances possibles du nœuds)
Démarrées avec Supervisord (et non avec tini)
➔plusieurs daemon sont démarrés

- sshd
- telegraf
- slurm (avec un choix selon le paramétrage docker-compose)
- munge



Quelques commandes SLURM

sinfo	interrogation des files d'attente
sbatch	soumission d'un job dans une file d'attente (appelées partitions dans SLURM)
salloc	réservation de ressources en interactif
srun	crée une allocation de ressources, à utiliser avec sbatch ou salloc run parallel jobs
scancel	suppression d'un job
squeue	liste des jobs dans les files d'attente
sprio	prorités relatives des jobs en attente
scontrol	affiche/modifie des données relatives aux tâches : jobs, nodes, partitions, reservations, etc.
sacct	affiche les données des jobs
sacctmgr	affiche et modifie les informations des comptes Slurm
sattach	s'attacher à une étape de travail en cours
sdiag	afficher les statistiques d'ordonnancement et les paramètres de synchronisation
sreport	rapports à partir des données de comptabilisation des travaux et des statistiques d'utilisation
sshare	afficher les parts et l'utilisation pour chaque compte de charge et chaque utilisateur
sstat	afficher les statistiques d'un travail ou d'une étape en cours d'exécution
sbcast	transmettre un fichier aux nœuds alloués à un travail Slurm.
scrontab	gestion de la crontab associée à slurm

TD élémentaire

INTERACTIF

Etapes (step0/1/2) de découvertes par rapport aux contraintes

On travaille dans le conteneur slurmctld

- on vérifie s'il y a une partition pour root

```
sacctmgr show association -p user=root
```

`scontrol show partition`

`scontrol show nodes`

`sinfo -Nel`

- Sinon

```
sacctmgr --immediate add cluster name=linux
```

  - Puis un restart des images

- Dans le répertoire /usr/local/var/mpishare faire
- git clone http://gogs.eldarsoft.com/M2_IHPS/glcs_slurm.git

Dans les répertoires test1/test2

```
mpicc -g3 -o elementary elementary.c
```

TD d'exploration

INTERACTIF

Etapes du TD

- Explorer les répertoires step0, step1, step2

`salloc -n 2 mpirun ./elementary` ➔ attention il faut que le binaire soit visible des nœuds (pb de partage du binaire – modifier les scripts…)

`sbatch script0.sh`

`sacct -j %jobid obtenu au moment du sbatch%`

- Dans le répertoire step0, modifier le script pour avoir un code retour différent de 0
- Vérifier le résultat du batch

`sbatch -n 2 xxx.sh` //(selon le répertoire)

`squeue -s -j %jobid%`

`sacct -j%jobid%`

`squeue -s -i 30 -j %jobid%`

`sacct -j %jobid%`

Test de sbatch

INTERACTIF

- sbatch
  - -N, --nodes=‹minnodes›[-maxnodes]|‹size_string›
  - Request that a minimum of minnodes nodes be allocated to this job.
  - -n, --ntasks=‹number›
  - sbatch does not launch tasks, it requests an allocation of resources and submits a batch script. This option advises the Slurm controller that job steps run within the allocation will launch a maximum of number tasks and to provide for sufficient resources. The default is one task per node, but note that the --cpus-per-task option will change this default.

```
sbatch -n2 -N2 -exclusive xxx.sh
squeue -O jobid,state,qos,timeused
```

Idée du TD – le problème du temps partagé

INTERACTIF

- SLURM « envoie » un signal SIGTERM avant la limite du temps du batch
- On veut observer la fin du batch ➔ raison du répertoire step2

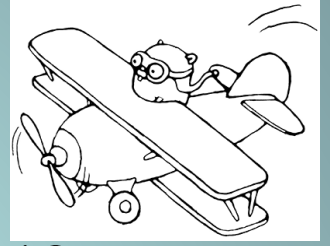
```
sacctmgr modify qos normal set MaxWall=00:00:02
```

Quelques dates pour le langage Go

- 2007 : Robert Griesemer, Rob Pike, Ken Thompson démarrent le projet d'un nouveau langage (Pike et Thomson sont co-auteurs du B –avant le C- et de l'UTF-8)
- 2009 : projet opensource publique
- mars 2012 : Go 1.0
- 12 janvier 2016 : une nouvelle incroyable !

"All Systems z are Go: IBM ports Google language to mainframes"

Le langage Go



- La création des langages est une activité continue
  50 nouveaux langages tous 10 ans
  (source https://en.wikipedia.org/wiki/Timeline_of_programming_languages)

Un langage est une représentation de l'espace du problème

Un langage n'est pas universel (tour de Babel) →idée de spécialisation (en 1996, 500 langages spécialisés)

(source http://www.cs.bsu.edu/homepages/dmz/cs697/langtbl.htm)

Il faut environ 7 ans pour qu'un langage soit populaire

PHP4.2 (2002) →2009

Python2.3 (2003)→2010

Go1.x(2009)→2016 //langage de l'année sur tiobe.com

Un langage "à la mode apparait" en fonction des pbs à "la mode"

Typologie d'un langage

- Les paradigmes sous jacents au delà du pure langage
  - Tout est liste (lisp, ruby, smalltalk) et rien d'autre
    - Smalltalk conceptualise l'objet à travers une vision organisée
  - Tout est objet (C++) y compris le clavier
    - C++ reprend le C et branche des objets sur tout (header, opérateur, template, …)
  - Tout est règle (SQL, Prolog)

- L'abstraction dans l'abstraction :
  - mettre une VM
  - mettre un ramasse miette
  - mettre des références à la place des pointeurs

Le langage Go

- break, default, func, interface, select, case, defer, go, map, struct, chan, else, goto, package, switch, const, fallthrough, if, range, type, continue, for, import, return, var

- 2009 : Go 1.0 – avec un compilateur
- **Go : 25 mots clés / pour ANSI C : 32 / pour Brainfuck : 8**
- **Approche CSP → les mêmes pbs que ceux du HPC**
- Langage Opensource
  - →Licence du langage : type BSD

- Aujourd'hui 3 compilateurs : llgo (llvm), gcc, gc
- Processeurs cibles

ARM, ARM64, x86-32 et AMD64, Mips32, Mips64, s390, PPC64, RiscV, loongarch64

- Typage et gestion des chaines de texte
  - Postfixé // gestion de l'utf-8 – Rob Pike & K. Thompson

```
var b rune = 'a'
var kanji_r rune = '門' // kanji pour dire porte E99680
var kanji_s string = "\xE9\x96\x80"
fmt.Printf("%c \n%s et le charactère a, ascii = %+v\n",
  kanji_r, kanji_s, b)
```

https://play.golang.org/p/XEVoMmK5mXw

Les variables

- Les variables et les constantes

```
var i = 'a'
i:='a'
const a int = 10
const (
    Red = (1<<iota)
    Green = (1<<iota)
    Blue,ColorMask (1<<iota,(1<<(iota+1))-1) )
```

→ramasse-miette

→référence &r, *r

- tableaux

itérateur

```
var s =[]string{2:"aa","bb"}
for a,c :=range s
  {     fmt.Printf("%d,%s\n",a,c)      }
```

```
0,
1,
2,aa
3,bb
```

- … = variadic

Arrays / Slices

- Slices ➔ un type plus intéressant

```
var s =[]string{2:"aa","bb"}
var t=[]string{4:"cc","dd"}
s=append(s,t...)
for a,c :=range s {     fmt.Printf("%d,%s\n",a,c)     }
```

```
0,
1,
2,aa
3,bb
4,
5,
6,
7,
8,cc
9,dd
```

https://play.golang.org/p/jXS5Rn0U8UR

Slices

```
var s =[]string{2:"aa","bb"}
var t=[]string{4:"cc","dd"}
s=append(s[2:4],t[4:6]...)
for a,c :=range s {     fmt.Printf("%d,%s\n",a,c)     }
```

```
0,aa
1,bb
2,cc
3,dd
```

https://play.golang.org/p/1eyKjt-3lkr

Allocation mémoire

- make()
- new()

  A:=new(int) // *A = 0

  →retourne un pointeur

  B:=make([]int,1) // B[0]=0

  →retourne une référence et fait l'allocation
- &LocalVar, &T{…}

  →retourne une référence

Switch

- Permet de choisir – plus élégant que l'accumulation de if

```
switch extension {
    case
      ".svg",
      ".png",
      ".pdf",
      ".tif",
      ".tiff",
      ".jpeg",
      ".jpg":
      return
    default : fmt.Printf("inconnu\n")
    }
```

https://play.golang.org/p/geaxSJGnYmR

boucles

- for i:=0; i<10;i++{
- for a,c := range {
  - break
  - continue (continue l'itération ou va à la condition d'arrêt)
  - goto

https://play.golang.org/p/61F0QcH3h4d

```
var t string =
    "0.6180339887498948482045868343656381177203091798057628621354486227052604628 18"
    for a,i := range t {
            fmt.Printf("a %d %c\n",a,i)
            if i == '9' {
                    goto Fin
            }
    }
Fin:
    fmt.Printf("fin")
```

https://play.golang.org/p/1J-Hzb0PAga

Les mots réservés du langage

- break
- default
- func
- interface
- select
- case
- defer
- go
- map
- struct
- chan
- else
- goto
- package
- switch
- const
- fallthrough
- if
- range
- type
- continue
- for
- import
- return
- var
- new
- make
- close
- ‹-
- …

- Le transtypage
  - Le langage est contraint
  - var x int = 300
  - var y int = 400
  - var mul float32
  - mul = float32(x) * float32(y)

  - On peut transtyper un int en byte puis en rune
  - rune(byte(val))

Création d'un objet : struct

```
type fenetre_2char struct {
        est_ascii bool
        val_ascii rune
}
const t string =
"0.6180339887498948482045868343656381177203091798057628621354486227052604628 18"
func test_struct() {
        sizeString := len(t) - 2
        for i := 2; i < sizeString; i = i + 2 {
                val, err := strconv.Atoi(t[i : i+2])
                if err == nil && val > 65 && val < 90 {
                        a_rune := []rune(t[i : i+2])
                        a := fenetre_2char{est_ascii: true, val_ascii: a_rune[0]}
                        fmt.Printf("iteration i: %d variable a %+v\n", i, a)
                }
        }
}
```

https://oeis.org/A001622

https://play.golang.org/p/AOqCFJRWERq

Pb! Quelle est la bonne conversion?

https://fr.wikibooks.org/wiki/Les_ASCII_de_0_%C3%A0_127/La_table_ASCII

Fonctions

Fonction

```
func (self fenetre_2char) Print() (rune, int) {
    fmt.Printf("Print() rune %c\n", self.val_ascii)
    a := byte(self.val_ascii)
    return self.val_ascii, int(a)
}
```

➔peut retourner plusieurs valeurs à la fois !

➔le receveur (=self) peut être une variable ou un pointeur sur une variable

Destructeur

- Mécanisme de finalisation // contrat sur le futur (quand l'objet est détruit)
- defer = finally en java = local destructeur en C++
- Permet de faire du code 'propre'
- Utile pour les DB, les filesystem, les IO (avec un close)

Application de defer

https://play.golang.org/p/4uo6CiZSINC

```
func test_struct() (total int) {
    var i int
    defer func() { total = i }()
    sizeString := len(t) - 2
    for i = 2; i < sizeString; i = i + 2 {
        val, err := strconv.Atoi(t[i : i+2])
        fmt.Printf("val %+v\n", val)
        if (err == nil) && (val > 65) && (val < 90) {
            a_rune := rune(byte(val))
            a := fenetre_2char{est_ascii: true, val_ascii: a_rune}
            fmt.Printf("iteration i: %d variable a %+v\n", i, a)
            a.Print()
        }
    }
    return
}
```

Comment faire sans le «defer » ?

Une fonction récursive en Python

```
def sum(k):
    def helper(n):
        if n == 0:
            return 0
        return n + helper(n-1)
    return helper(k)

def main():
    print(sum(3))

if __name__ == '__main__':
    main()  # print 6
```

La même fonction en Golang

```
package main
import (
    "fmt"
)
func sum(k int) int {
    helper := func(n int) int {
        if n == 0 {
            return 0
        }
        return n + sum(n-1)
    }(k)
    return helper
}
func main() {
    fmt.Println(sum(3))
}
```

https://play.golang.org/p/ba3O1Ev42yt

Ceci n'est pas du GO mais du C

```
var WeekDays = map[string]time.Weekday{"lundi": time.Monday, "monday": time.Monday,
    "mardi": time.Tuesday, "tuesday": time.Tuesday,
    "mercredi": time.Wednesday, "wednesday": time.Wednesday,
    "jeudi": time.Thursday, "thursday": time.Thursday,
    "vendredi": time.Friday, "friday": time.Friday,
    "samedi": time.Saturday, "saturday": time.Saturday,
    "dimanche": time.Sunday, "sunday": time.Sunday}

func DaysValidation(DaysList *[]string) error {
    for idx, day := range *DaysList {
        //          Cleaning input string
        (*DaysList)[idx] = strings.ToLower(strings.Replace(day, " ", "", -1))
        //          String verification
        _, is_valid := WeekDays[(*DaysList)[idx]]
        if !is_valid {
            return errors.New("Only English and French days are known")
        }
    }
    //      If all day are well writen, return nil error
    return nil
}
```

Ceci est du GO

```
var daysValidation = func() map[string]time.Weekday {
        return map[string]time.Weekday{"lundi": time.Monday,
                "monday":    time.Monday,
                "mardi":     time.Tuesday,
                "tuesday":   time.Tuesday,
                "mercredi":  time.Wednesday,
                "wednesday": time.Wednesday,
                "jeudi":     time.Thursday,
                "thursday":  time.Thursday,
                "vendredi":  time.Friday,
                "friday":    time.Friday,
                "samedi":    time.Saturday,
                "saturday":  time.Saturday,
                "dimanche":  time.Sunday,
                "sunday":    time.Sunday}
}
func printDay(key string) {
        a, err := daysValidation()[key]
        fmt.Println(err)
        fmt.Println(a)
}
func main() {
        printDay("lundi")
        printDay("londay")
}
```

https://play.golang.org/p/NYYyxDXaHmF

Ce qui nous reste à voir

- break
- default
- func
- interface
- select
- case
- defer
- go
- map
- struct
- chan
- else
- goto
- package
- switch
- const
- fallthrough
- if
- range
- type
- continue
- for
- import
- return
- var
- new
- make
- close
- ‹-
- …

fallthrough

https://play.golang.org/p/I2i0vZtItNP

```
func test_fallthrough() {
    i := 45
    switch {
    case i < 10:
        fmt.Println("i plus petit que 10")
        fallthrough
    case i < 50:
        fmt.Println("i plus petit que 50")
        fallthrough
        fmt.Println("bug")
    case i < 100:
        fmt.Println("i plus petit que 100")
    }
}
```

Fallthrough = cascade d'exécution

CSP : Communicating Sequential Process

- Fondamental dans les machines multicoeurs
- Notion de canal de liaison (channel) comme en MPI
- Permet de faire des actions 'en tâche de fond'

CSP – Aout 1978 – Tony Hoare

Programming Techniques
S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays
CR Categories: 4.20, 4.22, 4.32

grams, three basic constructs have received widespread recognition and use: A repetitive construct (e.g. the **while** loop), an alternative construct (e.g. the conditional **if..then..else**), and normal sequential program composition (often denoted by a semicolon). Less agreement has been reached about the design of other important program structures, and many suggestions have been made: Subroutines (Fortran), procedures (Algol 60 [15]), entries (PL/I), coroutines (UNIX [17]), classes (SIMULA 67 [5]), processes and monitors (Concurrent Pascal [2]), clusters (CLU [13]), forms (ALPHARD [19]), actors (Hewitt [1]).

The traditional stored program digital computer has been designed primarily for deterministic execution of a single sequential program. Where the desire for greater speed has led to the introduction of parallelism, every attempt has been made to disguise this fact from the programmer, either by hardware itself (as in the multiple function units of the CDC 6600) or by the software (as in an I/O control package, or a multiprogrammed operating system). However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor.

In order to use such a machine effectively on a single task, the component processors must be able to communicate and to synchronize with each other. Many methods of achieving this have been proposed. A widely adopted method of communication is by inspection and updating of a common store (as in Algol 68 [18], PL/I, and many machine codes). However, this can create severe problems in the construction of correct programs and it may lead to expense (e.g. crossbar switches) and

Les 7 propositions

- Executions parallèles de commandes avec démarrage synchrone et arrêt synchrone (au dernier élément)
- Echanges entre processus avec des E/S élémentaires via des **channels** bloquants
- Ceux-ci ont des actions déterminées en E/S sur l'état du processus
- Le concept de barrière non-déterministe
- Commande d'E/S avec barrière
- Commande d'E/S dans les boucles
- Capacité à choisir l'action en fonction du message

CSP : Communicating Sequential Process par Tony Hoare, 1978

- Communicating : les "channel" (chan)
  - **make** (chan string)
- Sequential process →objets concurrents (go)
  - **go** func
- Sequential process →blocage sur lecture/écriture dans une file // buffer
  - Symbole ‹-

Les «canaux» - channels

- chan : bidirectionnel
- chan <- : en écriture seulement
- <- chan : en lecture seulement

Hello World()

```
func Example() {
    ch := make(chan string)
    go func() { ch <- "Hello, world" }()
    fmt.Println(<-ch)
    // Output: Hello, world
}
```

https://play.golang.org/p/q-ugPxPk8d2

Les channels

https://play.golang.org/p/QxzMmS5QZXr

- Ils sont typés

```
func test_chan() {
    ch := make(chan int)
    fmt.Println("Sending value 1 to channel")
    go send(ch, 1)
    fmt.Println("Receiving from channel")
    go receive(ch)
    time.Sleep(time.Second * 1)
}
func send(ch chan int, i int) {
    ch <- i
}
func receive(ch chan int) {
    val := <-ch
    fmt.Printf("Value Received=%d in receive
    function\n", val)
}
```

Bufferisés

Buffered

https://play.golang.org/p/esBAGSAaJ5g

```
func test_buffered_chan() {
   ch := make(chan int, 1)
   ch <- 1
   fmt.Println("Sending value to channnel complete")
   val := <-ch
   fmt.Printf("Receiving Value from channel finished. Value received: %d\n", val)
}
```

https://play.golang.org/p/TYjoCF4D0HT

Chan en écriture

```
func write_example() {
    ch := make(chan int, 3)
    process(ch)
    fmt.Println(<-ch)
}
func process(chWrite chan<- int) {
    chWrite <- 2
    //s := <-chWrite
}
```

https://play.golang.org/p/yr8TU2hdgxY

Chan en lecture

```
func read_example() {
   ch := make(chan int, 3)
   ch <- 2
   process(ch)
   fmt.Println()
}
func process(chRead <-chan int) {
   s := <-chRead
   fmt.Println(s)
   //chRead <- 2
}
```

https://play.golang.org/p/ktkstsrmecB

Len() – effectif des msg, Cap() - capacité

```
func cap_len_example() {
    ch := make(chan int, 3)
    ch <- 5
    fmt.Printf("Len: %d, Cap %d\n\n", len(ch), cap(ch))
    ch <- 6
    fmt.Printf("Len: %d, Cap %d\n\n", len(ch), cap(ch))
    ch <- 7
    fmt.Printf("Len: %d, Cap %d\n\n", len(ch), cap(ch))
}
```

channel et opérateur range

https://play.golang.org/p/wvNlG5cax_z

```
func test_range() {
    ch := make(chan int)
    go sum(ch)
    ch <- 2
    ch <- 2
    ch <- 2
    close(ch)
    time.Sleep(time.Second * 1)
}
func sum(chRead <-chan int) {
    sum := 0
    for a := range chRead {
        sum += a
    }
    fmt.Printf("Sum: %d\n", sum)
}
```

Range permet de défiler un chan

Fermer un channel : close()

https://play.golang.org/p/Uvjas1wNQ61

```
func process_ch() {
    ch := make(chan int, 3)
    ch <- 2
    ch <- 2
    ch <- 2
    close(ch)
    sum(ch)
    time.Sleep(time.Second * 1)
}
func sum(ch chan int) {
    sum := 0
    for val := range ch {
        sum += val
    }
    fmt.Printf("Sum: %d\n", sum)
}
```

Fermeture du channel

On peut tester le statut du channel

https://play.golang.org/p/MNs0GPJeKkt

```
func test_status() {
  ch := make(chan int, 1)
  ch <- 2
  val, ok := <-ch
  fmt.Printf("Val: %d OK: %t\n", val, ok)
  close(ch)
  val, ok = <-ch
  fmt.Printf("Val: %d OK: %t\n", val, ok)
}
```

Résumé :

Commande	Réaction pour channel sans buffer	Réaction pour channel avec buffer	channel fermé	channel assigné à nil
chan <- **// écrire**	bloquant si pas de lecteur sinon succès	bloque si le channel est plein	panic	bloque indéfiniment
<- chan **// lire**	bloque s'il n'y a pas d'écrivain	bloque si le channel est vide	récupère le contenu du channel ou alors un type vide	bloque indéfiniment
len() //effectif des msg restant	o	effectif des msg restant	o pour les channel sans buffer – sinon effectif msg	o
cap() //capacité	o	capacité du channel	o pour les channel sans buffer – sinon capacité msg	o
close()	Succès	succès	panic	panic

Le select

https://play.golang.org/p/r92J8kn-rfE

```
const quit_value=999
func fibonacci(cWrite, cmdRead chan int) {
    fmt.Println("fibonacci started")
    x, y := 0, 1
    for {
        fmt.Printf("task fibo x = %d\n", x)
        select {
        case cWrite <- x:
            x, y = y, x+y
        case cmd_value := <-cmdRead:
            fmt.Printf("quit_value = %d\n", cmd_value)
            return
        //default : fmt.Printf("zzz\n")
}}}
func main() {
    c := make(chan int)
    q := make(chan int)
    go func() {
        fmt.Println("Goroutine started")
        for i := 0; i < 5; i++ {
            value := <-c
            fmt.Printf("main received %d\n", value)
        }
        q <- quit_value
    }()
    //close(c)
    fibonacci(c, q)}
```

Permet de filtrer le msg

Le select est bloquant, sauf avec **default**

Ce que nous avons vu

- break
- default
- func
- interface
- select
- case
- defer
- go
- map
- struct
- chan
- else
- goto
- package
- switch
- const
- fallthrough
- if
- range
- type
- continue
- for
- import
- return
- var
- new
- make
- close
- ‹-
- …

Les tests en langage Go

- Quels sont les avantages de prendre en charge les tests?
  - Les fichiers _test.go exposent les tests !
  - On conserve l'historique des tests (ça n'est plus dans le main)
  - Apporte de la documentation
  - Permet de faire de la non-régression
  - Permet de faire de la validation fonctionnelle (ie. driver)
  - ➔fait parti des LOC (lines of code)

Les tests

- Quels sont les tests ?
- Dans le langage Go, les tests sont pris en charge par le compilateur

- Pour tester un package (dans l'exemple c'est le package main):
  - Importer le package testing
  - Donner un nom de ficher en **_test.go**
  - Mettre des fonctions de signature Test[Majuscule](t* testing)
    - Par exemple : **func TestInsertHisto(t *testing.T) {**
- https://play.golang.org/p/oSDZYo0p0YC

Les tests

- Que décrit la variable t* testing ?

```
func TestMemoCds(t *testing.T) {
  var s memoCds
  s.lenCds = 1
  s.rawCds = "something"
  s.reset()
  fmt.Printf("s %+v\n", s)
  s.appendR('A')
  fmt.Printf("s %+v\n", s)
}
```

https://play.golang.org/p/2RbN69wBMmS

Les tests

- Objet t: 2 méthodes – Log,Error
- Un test est pensé pour une utilisation « automatique » ➔ l'idée d'un PASS/FAIL

```
func TestMemoCdsReset(t *testing.T) {
    var s memoCds
    s.lenCds = 1
    s.rawCds = "something"
    //s.reset()
    if (s != memoCds{}) {
        t.Error("s.reset doesn't work!")
    } else {
        t.Log("s.reset works")
    }
}
```

Les modules

- Présentation de go.mod = gestion des modules
- go mod init, go mod tidy

```
module ExtractCDS

go 1.16

require (
   configExtract v0.0.0-00010101000000-000000000000
   gonum.org/v1/plot v0.10.0
)

replace configExtract => ./configExtract
```

VERSION