Programmation C avancée

Concepts & Outils
pour le développement

maj 01/2023

cmake

- CMake est un outil simplifié permettant la compilation de sources C et C++.
- C'est un outil multi-plateformes sous licence BSD
- Nécessite la présence d'un fichier **CMakeLists.txt**

- gestion automatique des dépendances
- Simple d'utilisation / facile à prendre en main

cmake : hello world !

- Un seul fichier main.c que l'on souhaite compiler en un programme « hello_world » :

```
project(HelloWorld)
cmake_minimum_required(VERSION 3.0)

add_executable(hello_world main.c)
```

- CMakeList.txt peut être découpé sur plusieurs répertoires avec des inclusions
- Un projet <<HELLO>> avec une bibliothèque dans le répertoire Hello et un programme d'exemple dans le répertoire Demo

cmake

- ./CMakeLists.txt

```
cmake_minimum_required (VERSION 3.0)
project (HELLO)

add_library(hello hello.c)

add_executable (helloDemo demo.c demo_b.c)
target_link_libraries (helloDemo hello)
```

- Si je structure mon projet en deux répertoires: *hello* pour la bibliothèque et *demo* pour un exemple d’utilisation...

cmake

- ./CMakeLists.txt:

```
cmake_minimum_required (VERSION 3.0)
project (HELLO)

add_subdirectory (hello)
add_subdirectory (demo)
```

- ./hello/CMakeLists.txt

```
add_library (hello hello.c)
target_include_directories (hello PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

- ./demo/CMakeLists.txt

```
add_executable (helloDemo demo.c demo_b.c)
target_link_libraries (helloDemo LINK_PUBLIC hello)
```

cmake:cross platform make

- cmake est un système de compilation cross-plateformes. Il ne compile pas directement mais génère des fichiers dans différents formats :
  - Makefile
  - projet Visual Studio
  - Borland Makefile
  - projet Xcode
  - Kate
  - …
- cmake utilise les fichiers CMakeLists.txt et génère des fichiers en fonction de la plate-forme de compilation (Makefile, visual, xcode….).

cmake : les variables

- La déclaration de variables :
  - set(NAME VALUE)
  - ${NAME}
  - lors de l'appel à cmake : cmake -DNAME=VALUE
- Variables standards :
  - CMAKE_INCLUDE_PATH (pour les .h)
  - CMAKE_LIBRARY_PATH (pour la recherche de .so)
  - DESTDIR (pour l'installation)
  - CMAKE_BUILD_TYPE (Debug, Release)
- Dans le CMakeLists.txt :
  - CMAKE_C_FLAGS
  - CMAKE_C_FLAGS_DEBUG
  - CMAKE_C_FLAGS_RELEASE
  - CMAKE_CURRENT_SOURCE_DIR
  - CMAKE_CURRENT_BINARY_DIR
  - CMAKE_SOURCE_DIR

cmake : les fonctions

- add_executable(name sources)
- add_library(name STATIC sources)
- add_library(name SHARED sources)
- target_link_libraries(name libs)
- include_directories(dir1 dir2…)
- add_custom_command

cmake : utilisation

- cmake support l'out-source building : c'est à dire la compilation **en dehors** du répertoire des sources
- On suppose : projet/CMakeLists.txt
- alors on peut faire :

```
mkdir projet-build ; cd projet-build
cmake ../projet
make
```

- et

```
mkdir projet-debug ; cd projet-debug
cmake -DCMAKE_BUILD_TYPE=Debug ../projet
make
```

CMake: les modules

- Un module est un fichier écrit dans le langage de cmake avec l'extension: « .cmake »
- Les modules Find*Module*.cmake permettent de vérifier qu'un module est bien accessible et de positionner au besoin des variables.
- Par exemple « FindZLIB.cmake » vérifie que la bibliothèque libzlib.so et les headers sont bien disponibles sur le système. Des variables comme ZLIB_FOUND, ZLIB_LIBRARIES, ZLIB_INCLUDE_DIRS sont positionnées.

CMake: utilisation d’un module

- Dans un projet on écrit:

  find_package(ZLIB Required)

  include_directories(${ZLIB_INCLUDE_DIRS})

  ...

  target_link_libraries(hello ${ZLIB_LIBRARIES})

- Si zlib n’est pas trouvée, alors cmake s’arrêtera avec un message d’erreur lors de la phase de configuration.
- La commande *cmake --help-module-list* permet de lister l’ensemble des modules disponibles.

Question

#QDLE#Q#AB*#20#

- cmake analyse les dépendances :
  - A. en énumérant les fonctions appelées
  - B. en traçant les inclusions de fichiers

Question

#QDLE#Q#A*BCD#60#

- J'ai une bibliothèque dynamique libtoto.so compilée à partir de toto.c et toto.h. J'ai un programme de démo demo.c qui utilise cette bibliothèque. L'ensemble est compilé. Si je modifie toto.c je dois :
  - A. re-compiler la bibliothèque
  - B. re-compiler l'exécutable
  - C. réponse A & B
  - D. ne rien faire.

Question

#QDLE#Q#ABC*D#35#

- J'ai une bibliothèque dynamique libtoto.so compilée à partir de toto.c et toto.h. J'ai un programme de démo demo.c qui utilise cette bibliothèque. L'ensemble est compilé. Si je modifie toto.h je dois :
  - A. re-compiler la bibliothèque
  - B. re-compiler l'exécutable
  - C. réponse A & B
  - D. ne rien faire.

La gestion de sources

gestion de sources

- Lorsque l'on interagit avec d'autres développeurs, il est indispensable de pouvoir communiquer des propositions de modifications (ajout de fonctionnalité, correctif de bug, amélioration des perfs…) :

gestion de sources

#QDLE#S#ABC#25#

- Lorsque l'on interagit avec d'autres développeurs, il est indispensable de pouvoir communiquer des propositions de modifications (ajout de fonctionnalité, correctif de bug, amélioration des perfs…) :

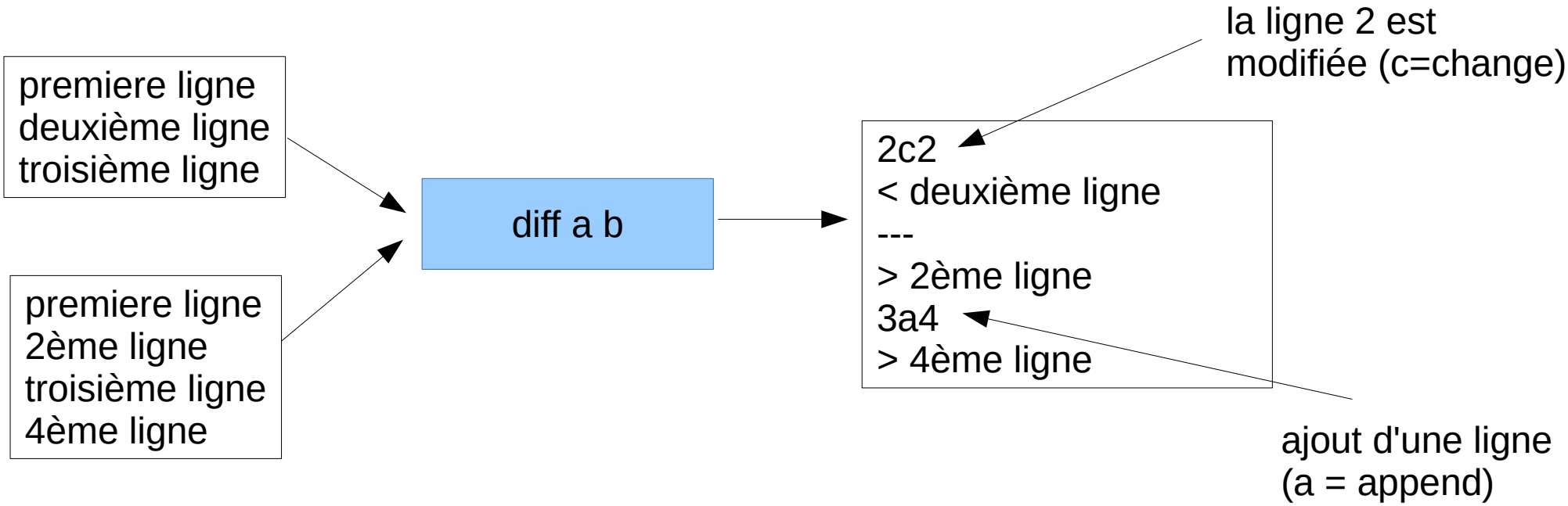
A) j'envoie tout le code en indiquant que c'est une nouvelle version.

B) j'écris un email détaillé des modifications à effectuer.

C) autre approche...

diff

- diff est un outil d'analyse de texte qui compare deux fichiers entre eux et produit le nombre minimum d'éditions de lignes à faire sur le premier fichier pour obtenir le second :



diff : side by side

- on peut afficher les deux fichiers cote à cote :

```
diff -y a b
```

```
premiere ligne                                  premiere ligne
deuxième ligne                                | 2ème ligne
troisième ligne                                 troisième ligne
                                              >     4ème ligne
```

- diff permet la comparaison récursive de deux arborescences.

diff : recursif

- Je souhaite modifier le code source d'un projet :
  1. je fais un copie de sauvegarde des sources d'origine
  2. j'effectue mes modifications
  3. à tout moment, je visualise mes modifications avec diff : `diff -r projet projet_new`

diff : recursif

- Je souhaite modifier le code source d'un projet :
  1. je fais un copie de sauvegarde des sources d'origine
  2. j'effectue mes modifications
  3. à tout moment, je visualise mes modifications avec diff : `diff -r projet projet_new`

```
diff -r GenTaskLib/TaskParser.cpp GenTaskLib_new/TaskParser.cpp
15c15
<         throw gtl::InvalidArgumentException("json parser error: expected a dictionnary at each task");
---
>         throw gtl::InvalidArgumentException("json parser error: expected a dictionnary for each task");
diff -r GenTaskLib/TaskParser.hpp GenTaskLib_new/TaskParser.hpp
36a37,38
>   /*! TaskParser: build a task from a json description.
>    */
```

Ajout d'un commentaire

diff : algorithme

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

  MOTELS ARE NOT HELL!

  MIROIR TU ES LA!

- Sur cet exemple « MO T » est une sous séquence commune.

diff : algorithme

#QDLE#Q#ABC*D#100#

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

  MOTELS ARE NOT HELL!

  MIROIR TU ES LA!

- Taille de la plus longue sous séquence commune ? (les espaces comptes).

  A. 6 B. 8 C. 9 D. 10

diff : algorithme

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

  MOTELS_ARE_NOT_HELL!

  MIROIR_TU_ES_LA!

diff : algorithme

- Chaque fichier est découpé en lignes
- Les lignes sont comparées entre elles
- Puis l'ensemble des lignes sont comparées entre elles (LCS où chaque « symbole » représente une ligne).

diff et communication

- Ainsi, si l'on souhaite communiquer une modification, il suffit d'envoyer le résultat d'un diff récursif : `diff -rupN original new > patch`
- On appel ce fichier un **patch**.
- Le destinataire peut lire ce fichier et comprendre vos modifications
- Il peut également appliquer ces modifications en local grâce au programme patch

- *les options upN sont nécessaire au fonctionnement de patch, elles ajoutent des informations de contexte (u), de fonction (p), d'ajout de fichier (N)*

patch

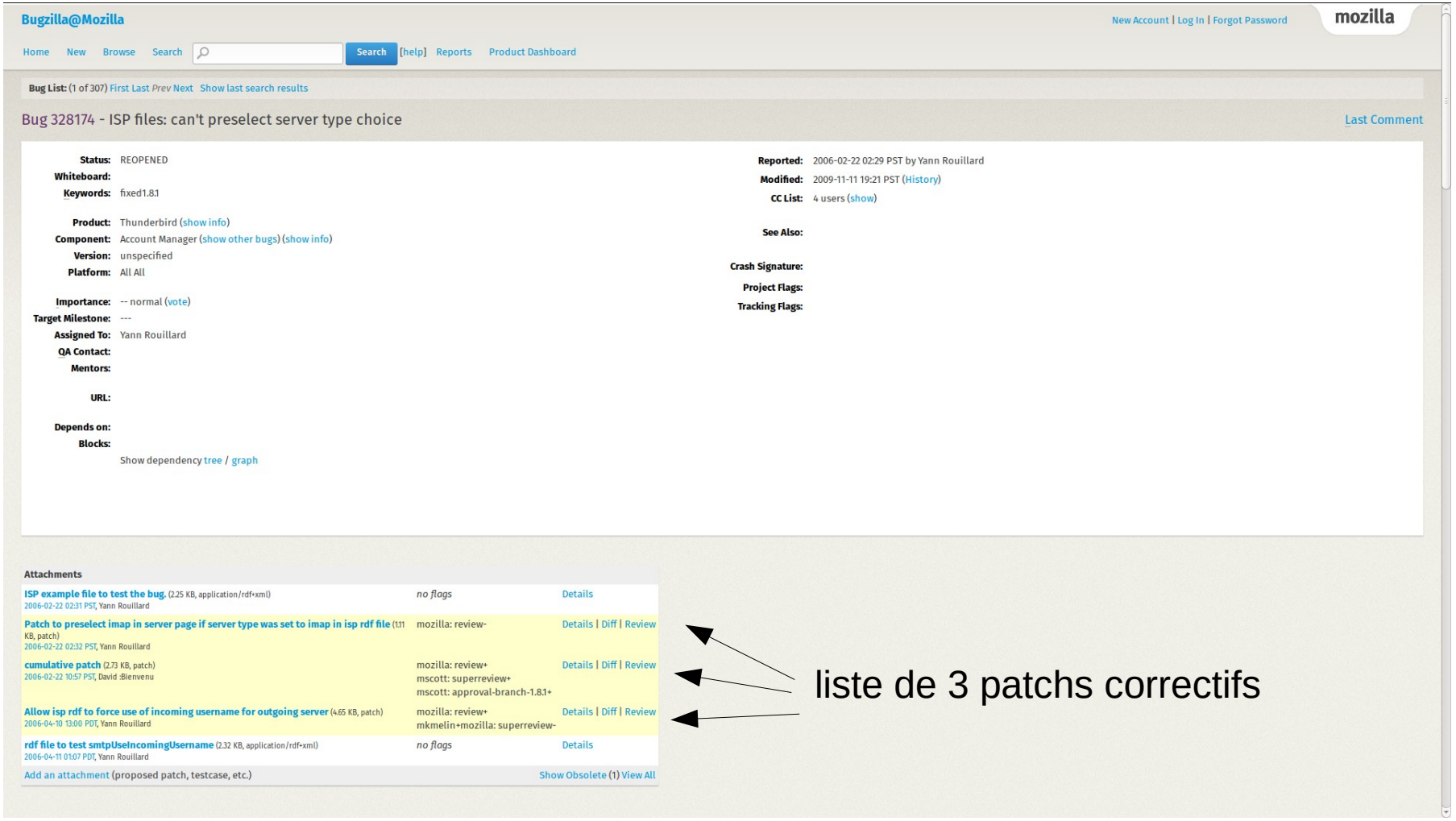
- le programme patch permet d'appliquer les modifications identifiées par diff.
- Ainsi sur l'exemple précédent je peux faire :

```
$ cp -R GenTaskLib GenTaskLibMod
$ cd GenTaskLibMod
$ patch < ../patch
patching file TaskParser.cpp
patching file TaskParser.hpp
$ cd .. ; diff -r GenTaskLib GenTaskLibMod
```

```
diff -r GenTaskLib/TaskParser.cpp GenTaskLibMod/TaskParser.cpp
15c15
<       throw gtl::InvalidArgumentException("json parser error: expected a dictionnary at each task");
---
>       throw gtl::InvalidArgumentException("json parser error: expected a dictionnary for each task");
diff -r GenTaskLib/TaskParser.hpp GenTaskLibMod/TaskParser.hpp
36a37,38
>   /*! TaskParser: build a task from a json description
```

diff & patch

- Dans le monde open source, diff et patch sont extrêmement utilisés



gestion de sources

- Un gestionnaire de sources permet de conserver l'historique des modifications apportées à un ensemble de fichiers.
- Il permet à un ensemble d'utilisateurs d’interagir sur un même code source.
- Tous les gestionnaires de code sources sont basés sur les principes de diff et patch
- Il existe deux grandes catégories de gestionnaires :
  - les centralisés
  - les décentralisés

Les gestionnaires centralisés

- Historiquement, cvs (concurent versioning system) et son successeur svn (subversion)
- Les gestionnaires centralisés reposent sur un serveur central qui archive toutes les modifications apportées au code.
- Chaque modification incrémente un numéro de révision
- Les commandes de base de svn :
  - checkout, update, infos, status, commit, diff, revert

svn

- On commence par la création d’un dépôt serveur : celui-ci n’est pas un espace de travail (on ne peut pas coder dedans).
- Chaque utilisateur interagit avec le serveur, il n'y a pas d'échange direct entre deux utilisateurs.
- Une méthodologie est associée à l'utilisation de svn, vous retrouverez cet méthodologie dans tout projet de développement (open source ou entreprise) => « svn red book »
- Cette méthodologie tente de répondre à des problèmes liés à tout développement logiciel (indépendamment de SVN).

svn : méthodologie

- Il est possible d'utiliser SVN juste pour le répertoire de développement principal.
- Seulement, comment faire si on a un « gros » développement à produire: si on transmet les modifications intermédiaires, le programme devient instable (ne compile plus par exemple...)
- Comment faire également pour gérer les versions:
  - On sort une version 1.0 qui évolue en 1.1 puis 1.2
  - On sort la version 2.0 (« casse » la compatibilité avec 1.x)
  - Un bug est trouvé dans la version 2.0: il faut le corriger dans la version courante mais également dans la version 1.x!

Un mot sur les versions...

- Il n'y a pas de « règles » universelles mais un certain nombre de pratiques.
- L'approche par numérotation est la plus répendue :

2.10.5

Major.Minor.Patch

Version.SousVersion.Patch

- Dans le cas d'une bibliothèque, on peut suivre les principes suivant :
  - Tant que l'API i.e. les fichiers d'entête et les fonctions (noms/signatures) ne changent pas, on garde le même numéro de version
  - On peut ajouter des fonctionalités : on incrémente le numéro de sous version
  - On peut corriger des bugs : on incrémente le numéro de patch

→ Un changement de version (Major) implique (sauf exception) une non compatibilité

Un mot sur les versions….

- Il existe d’autres stratégies de nommage/numérotation des versions.
- Par exemple, sur la base d’une date :
  - Ubuntu 18.04 => 2018 / Avril
- Parfois un mix :
  - Version.SousVersion.date(YYYYMMDD)
- On veille cependant à ce qu’il y ai un logique d’ordre
- Il est aussi apprécié que la logique de numérotation suit un ordre (lexicographique et/ou numérique).

Svn red book : les problèmes

- Les problèmes à résoudre sont :
  - Je dois pouvoir consulter les sources correspondantes à une version
  - Je dois pouvoir figer le développement en vue d’une mise en production sans empêcher les développement en cours
  - Je dois pouvoir développer une fonctionnalité sans empêcher d’autres développement tout en restant synchronisé avec ceux-ci
  - Je dois pouvoir intégrer ma fonctionnalité dans la version en cours de développement

svn : méthodologie

- La gestion des versions implique de maintenir plusieurs « répertoires » de développement en parallèle.
- Une méthodologie classique organise les sources ainsi:
  - / « racine »
  - /trunk : contient la version en cours des sources (dev.)
  - /branches/: des copies de trunk (« svn copy »)
  - /branches/1.0 : copie de trunk pour release (tests), retour de modif dans le trunk avec « svn merge » si compatible
  - /tags/1.0.0: version **figée** d'une branche qui est publiée, sert de référence. La branche correspondante est étiquetée (tag). **pas de commit/modifs dans ce répertoire**
  - /branches/modif_allali_155/: copie temporaire pour de « grosses » modifications avec retour dans le trunk par « merge ». Synchronisation régulière depuis le trunk (« merge »)

svn : la création d’un dépôt

- Le création d'un dépôt se fait à l'aide de la commande « svnadmin create nom_de_depot »
- possibilité d'ajouter l’exécution de script avant/pendant et après les « commits »
- possibilité d’envois de mails lors des commits
- facile à mettre en place sur son compte:
- cd ~/.depots/ ; svnadmin create SVN
- cd ~/; svn co file:///$HOME/.depots/SVN
- via ssh:
- svn co svn+ssh://allali@ssh.enseirb.fr/.depots/SVN

contrôle dans svn

- Il est possible d'avoir une approche hiérarchique dans svn en subdivisant le répertoire « branches » en répertoires et en ajustant les droits d’accès (lecture/ecriture) :
  - seuls les masters peuvent commiter dans « trunk »
  - les dev travaillent dans des branches
- Cela permet de faire de la revue de code

Les gestionnaires dé-centralisés

- Dans ce cas, il n'y a pas de dépôt central.
- Chaque utilisateur gère son propre dépôt.
- Un protocole permet l'échange de modifications (commit) entre deux utilisateurs.
- Exemple : git
- Commandes de base :
  - clone, add, commit, push, pull, fetch, merge, branch, checkout

git : les commits, pull et push

- Dans git, les commits sont locaux (il n'y a pas de dépôt central).
- On peut transmettre un ensemble (suite) de *commit* à un autre «dépôt» avec la commande *push*
- On peut réceptionner un ensemble de *commit* depuis un autre dépôt avec la commande *pull*.
- On peut ajouter/supprimer autant de dépôts connectés à notre dépôt avec « remote ». A faire dans les deux dépôts si l’on souhaite une symétrie.

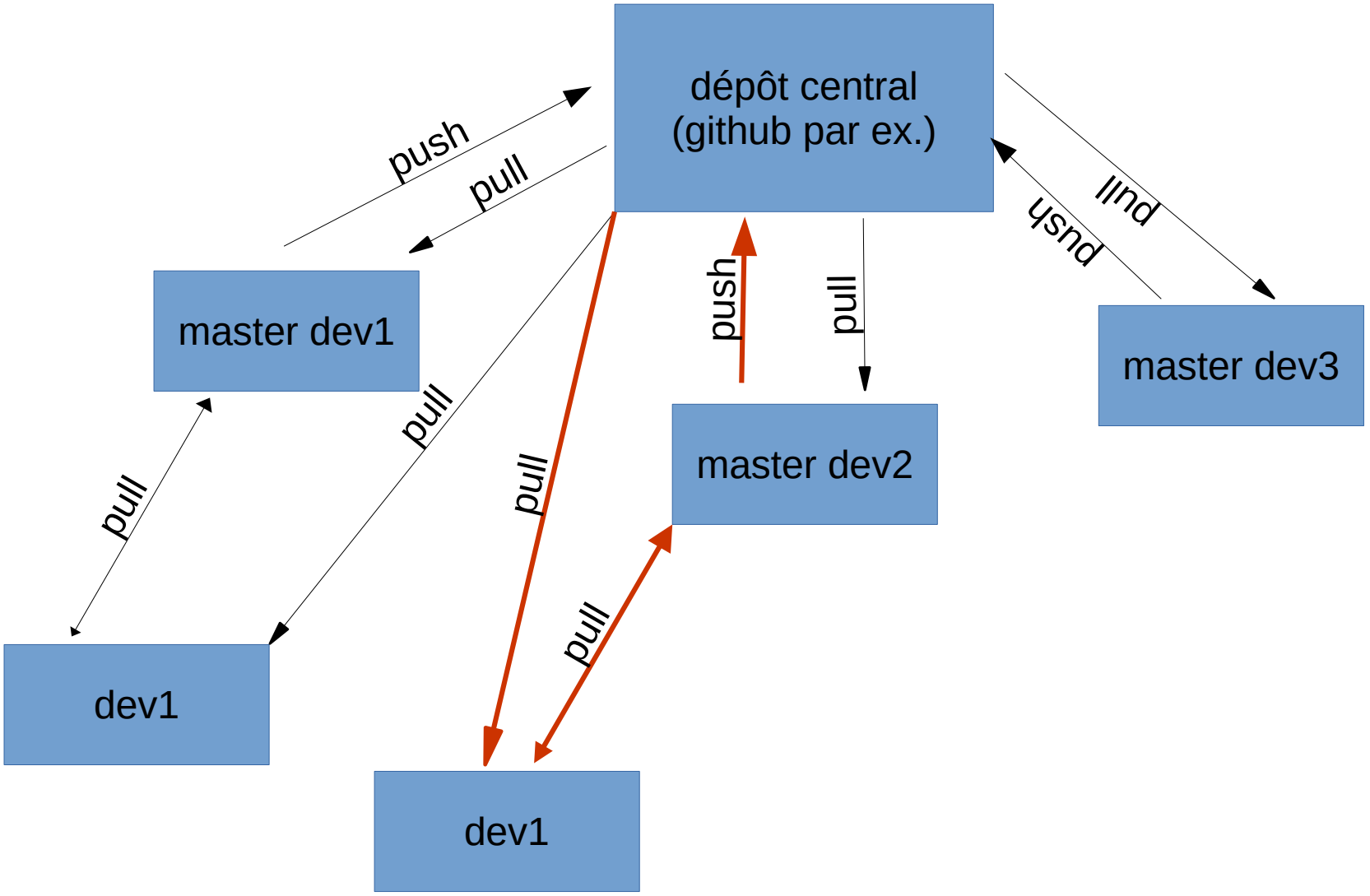
git

- git intègre une gestion native des branches (qui ne sont pas de répertoires mais des *vues/états*) :
  - création :
    - git branch b2
    - git checkout b2
    - ou bien git checkout -b b2
  - la fusion :
    - on bascule dans la branche qui doit recevoir les modifs
    - git checkout master ; git merge b2
    - Les modifications doivent avoir été *commitées* dans b2
  - la délétion : git branch -d b2
- La branche par défaut s'appelle master (paramétrable lors de la création d'un dépôt).

re-centralisation

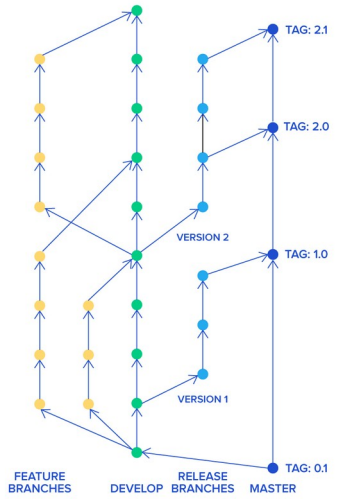
- L'avantage d'être en décentralisé et de pouvoir faire des « commit » sans connexion à un serveur.
- Pour la plupart des projets, il est cependant nécessaire d'avoir une référence : on utilise alors un dépôt git comme tel (github par exemple). Un tel dépôt s’appelle un « bare », il a la spécifité d’accepter les *push*
- On peut ensuite mettre en place un système de propagation hiérarchique des « commits » (des plateforme proposent un système de *pull request* pour cela).

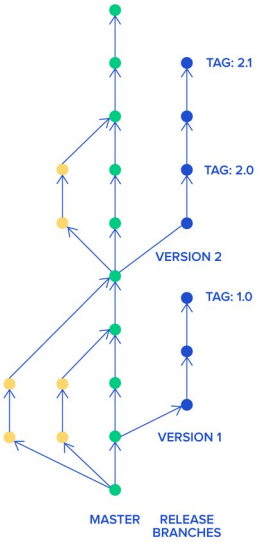
git


dépôt central
(github par ex.)
push
pull
master dev1
pull
pull
dev1
push
pull
master dev2
pull
dev1
llnd
ysnd
master dev3

git: méthodologie

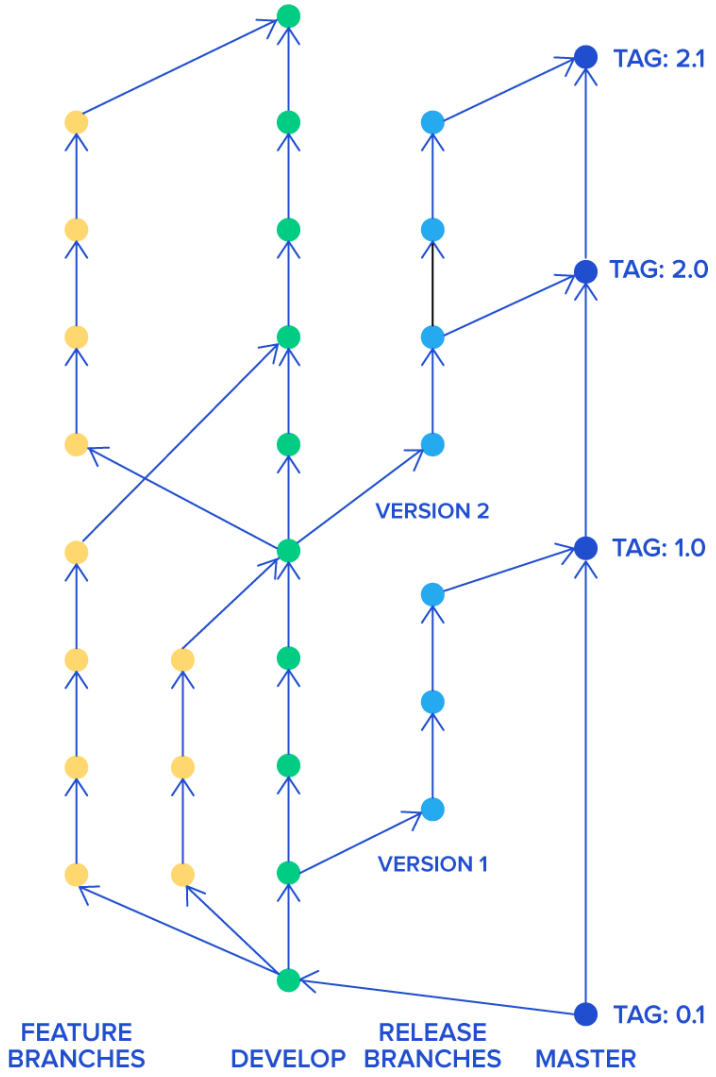
- L’utilisation de git repose nativement sur les branches
- Il y a deux approches actuellement:
  - git flow
  - trunk based





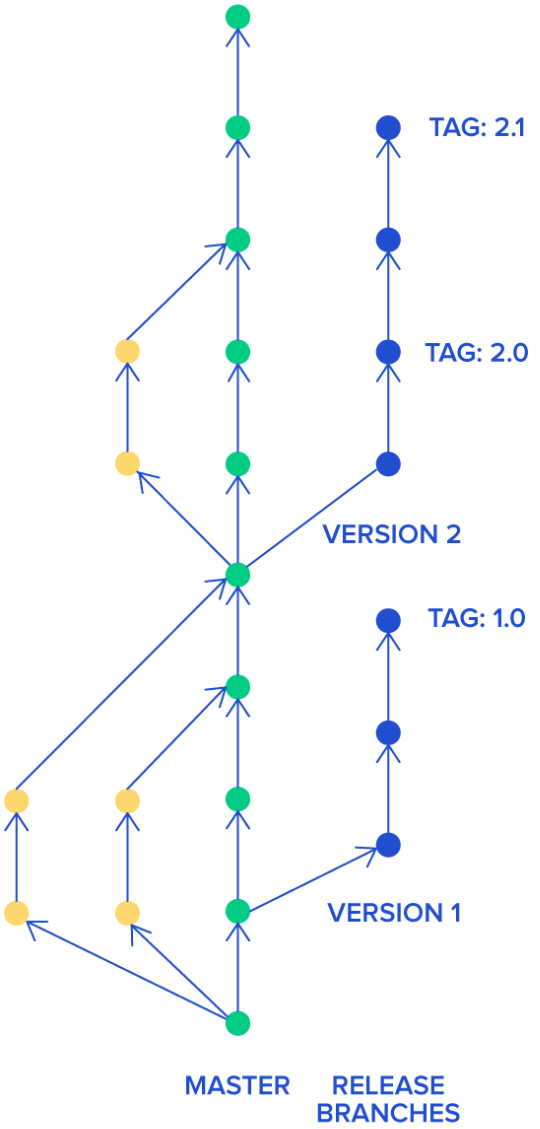
GitFlow

- *GitFlow* suppose des branches avec une durée de vie assez longue: il y a des aller-retour entre ces branches et une branches de développement (et les autres branches), elle même en synchronisation avec une branche de pré-release et la branche master de qui contient le code stable



Trunk Based

- *trunk based* privilégie les branches à durée de vie courte synchronisée avec master/trunk
- les release sont maintenues à coté.



GitFlow ou trunk based

- Les deux méthodologies ont leurs avantages et inconvénients
- L’environnement, le type de projet, le nombre de développeurs sont autant de facteurs qui plaident pour l’un ou l’autre des solutions

git ou svn?

- Différences majeures entre svn et git est :
  - centralisé / dé-centralisé
  - support natif du système de branches dans git
  - commit locaux dans git
- aujourd'hui git est de loin de le gestionnaire de sources le plus utilisé

Les autres gestionnaires

	Open Source	Centralisé	Décentralisé
CVS	●	●	
SVN	●	●	
GIT	●		●
SourceSafe		●	
Mercurial	●		●
Bazaar	●		●
BitKeeper			●
Team Foundation Server		●	

Il en existe bien d’autres (voir : https://en.wikipedia.org/wiki/List_of_version-control_software)

les plateformes

- De nombreuses plateformes proposent l'hébergement de dépôts telles que github et gitlab

  mais aussi bitbucket...

- ajout de services tels que l'intégration continue, suivi de bugs, wiki etc....