Travail en réseau et collaboratif sous Linux

Pierre Senellart

Semaine *Informatique pratique*, septembre 2026

Plan

La pile de protocoles d'Internet

Une pile de protocoles de communications.

Application	HTTP, FTP, SMTP, DNS, SSH	
Sécurité	SSL/TLS	(chiffrement, authentification)
Transport	TCP, UDP, QUIC	(fractionnement, fiabilité...)
Réseau	IP (v4, v6), ICMP	(routage, adressage)
Lien	Ethernet, 802.11 (ARP)	(adressage local)
Physique	Ethernet, 802.11 (physique)	

IP (Internet Protocol) [IETF(1981)]

- Adressage de machines et routage sur Internet
- Deux versions du protocole IP : IPv4 (historique) et IPv6
- IPv4 : adresses de 4 octets affectées à chaque ordinateur, p. ex., 137.194.2.24. Les institutions obtiennent des gammes de telles adresses, qu'elles assignent comme elles le souhaitent.
- Problème : seulement 2^{32} adresses possibles (en fait, beaucoup moins, beaucoup d'entre elles ne peuvent pas être utilisées, pour de nombreuses raisons). Beaucoup d'hôtes connectés à Internet n'ont donc pas d'adresse IPv4 propre et un système de network address translation (NAT) est utilisé.

IPv6 [IETF(2017)]

- IPv6 : adresses de 16 octets ; espace d'adressage beaucoup plus grand ! Les adresses ont la forme 2001:660:330f:2::18 (ce qui signifie 2001:0660:330f:0002:0000:0000:0000:0018). Autres fonctionnalités : multicast, autoconfiguration, etc.
- Déploiement inégal [Arcep(2025)] : très répandu en France côté accès grand public (94 % en fixe, 83 % en mobile), beaucoup moins dans les entreprises et côté serveurs (38 % des sites Web) ; une connexion entre deux machines quelconques passe donc encore le plus souvent par IPv4
- Une machine a en général plusieurs adresses (IPv4, IPv6, adresse de boucle locale 127.0.0.1 ou ::1, adresses privées derrière un NAT) : `ip addr` les liste

TCP (Transmission Control Protocol) [IETF(2022b)]

- L'un des deux protocoles principaux de transport utilisés au-dessus d'IP, avec UDP (User Datagram Protocol)
- Contrairement à UDP, fournit une garantie de transmission des données (acknowledgments)
- Les données sont divisées en de petits segments qui sont envoyés sur le réseau, et possiblement réordonnés à l'arrivée
- Comme UDP, chaque transmission TCP indique un numéro de port (entre 0 et $2^{16} - 1$) pour la source et la destination afin de distinguer d'autres flux de trafic entre les mêmes machines
- Un client utilise un numéro de port aléatoire pour établir une connexion à un numéro de port fixé sur le serveur
- Le numéro de port du serveur identifie conventionnellement le protocole d'application au-dessus de TCP/IP : 22 pour SSH, 80 pour HTTP, 443 pour HTTPS... voir `/etc/services` ; les numéros < 1024 sont privilégiés (seul le superutilisateur peut y démarrer un serveur)

DNS (Domain Name System) [IETF(1999)]

- Les adresses IPv4 sont difficiles à mémoriser, et un service donné (p. ex., le Web) peut changer d'adresse IP (p. ex., nouvel hébergeur ; voire changements d'IP fréquents en cas de pénurie d'adresses IPv4)
- Encore plus difficile de retenir des adresses IPv6 !
- DNS : un protocole (principalement sur UDP, mais aussi sur TCP, et désormais chiffré : DoT, DoH) qui associe des noms lisibles (p. ex., `www.google.com`, `weather.yahoo.com`) à des adresses IP
- Nom de domaine hiérarchique : `com` est un domaine de plus haut niveau (TLD), `yahoo.com` un domaine de deuxième niveau, etc.
- Résolution hiérarchique des noms de domaines : serveurs racines avec des adresses IP fixes qui connaissent les serveurs en charge d'un TLD, qui connaissent les serveurs en charge des sous-domaines, etc.
- Rien de magique avec `www.google.com` : juste un sous-domaine de `google.com`.

Internet et le Web

Internet : réseau physique d'ordinateurs (ou hôtes)

World Wide Web, Web, WWW : collection logique de documents hyperliés

- contenu statique et dynamique
- Web public et Web privé
- chaque document (ou page Web, ou ressource) identifié par une URL

URL (Uniform Resource Locator) [IETF(2005)]

`https` `://` `www.example.com` `:443` `/` `path/to/doc` `?name=foo&town=bar` `#para`

schéma — hôte — port — chemin — requête — fragment

- **schéma :** manière d'accéder à la ressource, généralement http ou https
- **nom d'hôte :** nom de domaine de l'hôte (cf. DNS) ; possibilité d'hôtes virtuels.
- **port :** port TCP ; par défaut 80 pour `http` et 443 pour `https`
- **chemin :** chemin logique de la ressource
- **requête :** paramètres additionnels optionnels
- **fragment :** sous-partie optionnelle de la ressource

URL relatives à un contexte (p. ex., l'URL ci-dessus) :

`/titi`	`https://www.example.com/titi`
`tata`	`https://www.example.com/path/to/tata`

HTTP (HyperText Transfer Protocol) [IETF(2022a)]

- Protocole d'application à la base du Web
- Diverses versions cohabitent : HTTP/1.1 (messages en mode texte), HTTP/2 et HTTP/3 (messages binaires ; HTTP/3 s'appuie sur QUIC, donc UDP et non TCP)

- Requête du client :

```
GET /MarkUp/ HTTP/1.1
Host: www.w3.org
```

- Réponse du serveur :

```
HTTP/1.1 200 OK
...
Content-Type: text/html; charset=utf-8

<!DOCTYPE html ...> ...
```

- Deux méthodes HTTP principales : GET et POST ; des en-têtes additionnels, dans la requête et la réponse
- En ligne de commande : `curl -v` *URL* montre requête et réponse ; `wget` pour télécharger

Plan

Principes de base

- Les connexions réseaux se font via une série d'appels systèmes :

  `socket` pour créer un connecteur réseau

  `bind` pour associer la socket à une interface réseau (serveur)

  `listen` pour écouter sur un port (serveur)

  `accept` pour accepter les connexions (serveur)

  `connect` pour établir une connexion (client)

  `select`, `read`, `write`, `poll` pour lire, écrire, attendre des messages, etc.

- Les sockets ouvertes apparaissent comme des fichiers ouverts dans `/proc` (voir aussi `lsof -i`)
- Utiliser `strace` pour analyser les connexions faites par un processus !

`ss` (et `netstat`)

- Affiche l'ensemble des sockets ouvertes : connexions en cours et serveurs en attente de connexion
- `netstat` est l'outil historique (paquet net-tools, plus maintenu) ; `ss` (iproute2) le remplace avec les mêmes options
- On peut sélectionner le type de socket :
  - `-t` pour TCP
  - `-u` pour UDP
  - `-x` pour Unix
- On obtient avec `-p` les processus en question
- On peut sélectionner les connexions en cours (par défaut), les serveurs en attente (`-l`) ou toutes (`-a`)

nc et socat

- `nc` : Outil léger et très pratique pour établir des connexions
- `nc -l` *p* : serveur TCP (ou UDP avec `-u`, socket Unix avec `-U`) sur le port *p*
- `nc` *h p* : client TCP (ou UDP, ou Unix) vers l'hôte *h*, port *p*
- Attention, deux variantes incompatibles : `nc -l` *p* (OpenBSD, la plus courante) mais `nc -l -p` *p* (netcat traditionnel)
- Pour des besoins plus avancés (connexions multiples au même serveur, connecter deux serveurs, chiffrement, etc.), on peut utiliser `socat` :
  - `socat TCP4-LISTEN:p -`
  - `socat - TCP:h:p`

Connexion SSL/TLS

- Beaucoup de protocoles modernes utilisent une couche de chiffrement (HTTPS, SMTPS, etc.)
- Pour établir une telle connexion TCP vers hôte *h* port *p* :

```
openssl s_client -crlf -connect h:p
```

`dig` : client DNS très complet

- Permet d'effectuer des requêtes DNS arbitraires
- On peut spécifier un type d'entrée DNS (`-t`), telle que A, AAAA, CNAME, NS, MX (ou AXFR pour un transfert de zone complet, sur TCP, si le serveur l'autorise)
- On peut spécifier à quel serveur DNS on s'adresse avec `@serveur` ; `+trace` suit la résolution depuis les serveurs racines
- Pour des requêtes plus simples : `host`

Capture du trafic réseau

tcpdump capture l'ensemble du trafic réseau local (pas seulement TCP) – très puissant ! nécessite en général les droits de superutilisateur

wireshark interface graphique permettant d'analyser ces résultats

mitmproxy proxy HTTP(S) pour capturer le trafic réseau entre l'ordinateur local et un site Web, en configurant l'utilisation de ce proxy

au sein d'un navigateur utiliser les outils de développement (onglet *Réseau*), qui indiquent le trafic réseau réalisé par le navigateur

Mais aussi...

`ip` pour les paramètres réseaux locaux (interfaces réseaux, adresses IP, routage, etc.) ; `ip neigh` pour la table ARP locale (anciennement `arp -a`)

`ping` pour envoyer une requête ICMP echo vers une machine, manière la plus légère de tester son existence (mais elle peut ne pas répondre) et d'estimer la latence de la connexion

`traceroute` pour envoyer des paquets (UDP par défaut sous Linux, ICMP avec `-I`) avec des durées de vie (TTL) croissantes vers une machine afin d'estimer la topologie du réseau ; `mtr` combine ping et traceroute

`whois` pour accéder à la base Whois des informations sur les propriétaires de noms de domaine ou d'adresse IP

`iftop` pour visualiser le trafic réseau en temps réel

Plan

Secure Shell

- Protocole (et outils associés) permettant de se connecter à distance à un ordinateur en mode texte
- La connexion est chiffrée, l'authentification (par mot de passe ou par clef cryptographique) est gérée par SSH
- Une fois la connexion établie, SSH lance le shell de login de l'utilisateur
- Usages : utilisation d'un serveur de calcul, connexion à une machine distante, travail depuis un ordinateur sans installation Linux locale, accès à un dépôt git (GitHub, GitLab), etc.
- Un serveur SSH doit être installé sur la machine cible

Usage de base

- ssh login@machine pour se connecter à `machine` en tant que `login`
- À la première connexion, SSH affiche l'empreinte de la clef de la machine et demande confirmation ; elle est ensuite mémorisée dans `$HOME/.ssh/known_hosts` et toute modification déclenche une alerte
- Le fichier $HOME/.ssh/config est un fichier de configuration dans lequel on peut lister des alias de machines, des noms d'utilisateurs pour ces machines, des options de SSH, etc.
- Pour terminer la connexion, terminer le shell
- Possible de juste lancer une commande sans lancer le shell : `ssh login@machine commande` (ajouter `-t` si besoin d'un pseudo-terminal)
- Si connexion bloquée (problème réseau), interrompre la connexion avec « Entrée + ∼ + . »

Clef SSH

- Plutôt que de s'authentifier par mot de passe, on peut répondre à un challenge cryptographique prouvant qu'on détient la clef privée associée à une clef publique communiquée au serveur
- Nombreux avantages : pas besoin d'accès par mot de passe sur la machine, clef toujours accessible ou protégée par mot de passe mais déverrouillée une fois par session de travail, clef transmissible de serveur en serveur, etc.
- Pour créer une clef, `ssh-keygen` : génère une clef publique (`id_ed25519.pub`) et une clef privée (`id_ed25519`) dans `$HOME/.ssh/` ; mot de passe de protection de la clef privée choisi à la génération (l'algorithme ed25519 est le défaut depuis OpenSSH 9.5, les clefs RSA `id_rsa` restent courantes)

Installer sa clef

- La clef publique doit être mise sur le serveur, dans `$HOME/.ssh/authorized_keys` (plusieurs clefs possibles, une ligne par clef) ; `ssh-copy-id login@machine` le fait pour vous
- C'est aussi cette clef publique que l'on enregistre sur GitHub ou GitLab pour y accéder en SSH
- La clef privée ne quitte jamais votre machine : ne la copiez pas sur un serveur, ne la partagez pas

Agent SSH

- Un agent SSH est un logiciel tournant sur la machine cliente et gardant en mémoire la clef privée déverrouillée
- Souvent installé par défaut dans les distributions modernes ; peut être lancé à la main comme parent d'un shell avec `ssh-agent zsh` (sera accessible depuis ce shell)
- `ssh-add -l` pour lister les clefs en mémoire
- Clefs automatiquement ajoutées à l'agent à la première connexion si l'option de configuration `AddKeysToAgent` est positionnée à `yes` (dans `$HOME/.ssh/config`)
- Cette clef peut être transmise par l'agent au travers d'une connexion SSH pour être utilisée pour se connecter à une machine C via une machine B depuis une machine A détenant la clef ; option de configuration `ForwardAgent` positionnée à `yes`. Attention : un administrateur malveillant de B peut alors utiliser la clef ; préférer le rebond ci-dessous

Rebond

- Souvent impossible de se connecter directement en SSH à la machine C qui nous intéresse (par exemple, derrière un pare-feu) ; une connexion initiale via une machine B est nécessaire
- En ligne de commande : `ssh -J B C`
- Ou, une fois pour toutes, dans `$HOME/.ssh/config` :

```
Host C
ProxyJump B
```

- Alors, `ssh login@C` passera automatiquement par `B` ; si un agent est bien configuré, pas de mot de passe demandé ! La connexion vers C est chiffrée de bout en bout, B ne voit rien passer
- (Ancienne syntaxe, avant OpenSSH 7.3 : `ProxyCommand ssh B nc %h %p`)

Redirection de ports

- SSH peut transporter n'importe quelle connexion TCP dans le tunnel chiffré
- `ssh -L 8888:localhost:8888 B` : le port 8888 de la machine locale est redirigé vers le port 8888 *vu depuis B* (ici, B lui-même) ; cas typique : un carnet Jupyter ou une interface Web lancés sur un serveur de calcul, consultés dans le navigateur local sur `http://localhost:8888/`
- Plus généralement `-L port_local:hôte:port` : `hôte` est résolu depuis B, ce qui donne accès aux machines du réseau de B
- `ssh -R` : l'inverse, un port de B redirigé vers la machine locale
- `-N` pour n'ouvrir que le tunnel, sans shell ; options `LocalForward` et `RemoteForward` dans la configuration

Proxy SOCKS

- SSH peut servir à établir un proxy SOCKS, qui permet de faire transiter le trafic (par exemple Web, via la configuration du proxy du navigateur) d'une machine A à une machine W via une machine B (comportement similaire à un VPN)
- Lancer `ssh -D 12345 B` pour démarrer le serveur SOCKS
- Configurer sur la machine locale le navigateur (ou autre logiciel) pour utiliser le proxy SOCKS sur la machine locale (`localhost`) et sur le port 12345
- Le trafic Web sera alors vu comme venant de la machine B !

Affichage graphique

- SSH peut aussi faire en sorte que des programmes graphiques X11 lancés sur la machine distante apparaissent sur la machine locale (on parle d'export X11)
- Lancer avec `ssh -X` ou option de configuration `ForwardX11`
- Peut dépanner, mais assez lent
- Alternative : VNC, protocole de connexion graphique à distance, divers serveurs et clients
- Souvent inutile : les éditeurs modernes savent éditer des fichiers distants au travers de SSH (TRAMP pour emacs, Remote – SSH pour VS Code) ou tournent dans le terminal (vim)

Transfert de fichiers

- SSH permet aussi le transfert de gros volumes de données, telles que des fichiers
- Diverses variantes et outils : `scp`, `sftp`, etc.
- Recommandé : rsync, un outil de transfert (et synchronisation) d'une hiérarchie de répertoires d'une machine à une autre – utilise SSH (et les alias, la configuration) par défaut
- `rsync -avzP repertoire serveur:chemin` transfère de machine locale au serveur
- `rsync -avzP serveur:chemin repertoire` transfère du serveur à la machine locale
- Options : `-a` transfère récursivement, en préservant quasiment tout ce qu'il est possible de préserver ; `-v` affiche chacun des fichiers transférés ; `-z` compresse à la volée pour gagner du temps ; `-P` permet le transfert en plusieurs fois de gros fichiers
- `sshfs` monte un répertoire distant comme s'il était local

Depuis une machine non Linux

- Clients SSH pour de nombreux systèmes :
  - **Windows** putty (et WinSCP pour transfert de fichier), souvent plus pratiques ; OpenSSH est aussi intégré depuis Windows 10 (`ssh` dans un terminal), et WSL fournit un Linux complet
  - **Android** ConnectBot (et AndFTP pour transfert de fichier) ou Termux (terminal et environnement Linux traditionnel sous Android)
  - **macOS, autres Unix** ssh en ligne de commande
- Pour l'export X11, il faut un serveur X11, tel que Xming ou VcXsrv pour Windows

Plan

Multiplexeurs de terminaux

- Logiciels tournant au sein d'un terminal, qui sont eux-mêmes des émulateurs de terminaux
- Serveur en arrière plan qui maintient une collection de pseudo-terminaux, avec des processus lancés dans chacun
- Client permettant à tout moment de récupérer l'accès à chacun de ces terminaux (s'attacher), ou d'en redonner le contrôle au serveur (se détacher)
- Permet en particulier (mais pas seulement !) de lancer des commandes potentiellement interactives dans un terminal et de se déconnecter de la machine sans que ces commandes perdent leur terminal
- Idéal pour du travail à distance
- Plein d'autres fonctionnalités, en particulier gestion de fenêtres multiples

`screen`

- Multiplexeur de terminal historique, présent sur beaucoup de systèmes
- Créer une session : `screen` (on peut lui donner un nom avec `-S`)
- Lister les sessions : `screen -ls`
- S'attacher à une session : `screen -r` suivi du PID ou du nom de la session (avec `-d`, on détache une session éventuellement en cours ; avec `-x`, on peut s'attacher à une session déjà attachée)
- Se détacher d'une session : CTRL+a d

`tmux`

- Multiplexeur de terminal plus moderne que `screen` avec interface plus agréable
- Créer une session : `tmux new` (on peut lui donner un nom avec `-s`)
- Lister les sessions : `tmux ls`
- S'attacher à une session : `tmux attach -t` suivi du nom ou du numéro de la session (avec `-d`, on détache une session éventuellement en cours ; on peut s'attacher à une session déjà attachée)
- Se détacher d'une session : CTRL+b d

Plan

Le problème

- Programmer produit de nombreuses versions des mêmes fichiers
- Solutions artisanales typiques :
  - ne garder que la dernière version
  - numéroter à la main : `projet_final`, `projet_final2`, `projet_final_vraiment`
  - copier des répertoires pour les partager
- Ces approches échouent quand :
  - on veut revenir en arrière
  - on travaille à plusieurs, avec des contributions plus ou moins indépendantes
  - on introduit un bug sans savoir quand
  - on veut une sauvegarde automatique
- On parle ici de programmation, mais le problème est le même pour toute activité créative sur ordinateur : rapports, articles, thèses (en LaTeX !), conception assistée, illustration, animation

Qu'est-ce que le contrôle de versions ?

- Un système de contrôle de versions (VCS, *version control system*) :
  - enregistre l'évolution des fichiers au cours du temps
  - conserve un historique structuré des modifications
  - permet de retrouver un état antérieur
  - facilite la collaboration
- Chaque modification est :
  - attribuée à un auteur
  - datée
  - décrite par un message
- Un outil fondamental pour quiconque programme

Vocabulaire

- Dépôt (*repository*)
  - Un ensemble de fichiers et tout leur historique
  - Local, ou hébergé à distance (p. ex., GitHub, GitLab)
- Commit
  - Un instantané du dépôt à un moment donné, une version précise
  - Identifié par un identifiant unique, accompagné de métadonnées (message, auteur, etc.)
- Branche
  - Une ligne de développement nommée dans le dépôt
  - Permet une évolution indépendante du code
- Clone
  - Une copie locale d'un dépôt
  - Contient tous les fichiers et l'historique complet

Un peu d'histoire

- Premiers systèmes (années 1980–1990) :
  - RCS, CVS
  - modèle centralisé
  - branches et fusions mal prises en charge
- Deuxième génération :
  - Subversion (SVN)
  - toujours centralisé, mais plus efficace, robuste, souple
- Systèmes modernes :
  - systèmes de contrôle de versions distribués (DVCS)
  - git, Mercurial

Centralisé ou distribué

- VCS centralisé (CVS, SVN) :
  - un serveur central
  - chaque utilisateur a une copie de travail locale et informe le serveur de chaque modification
  - l'historique est sur le serveur
- VCS distribué (git, Mercurial) :
  - chaque copie de travail contient l'historique complet
  - les commits sont locaux
  - la collaboration consiste à échanger des commits
- Conséquences :
  - on peut travailler hors ligne
  - pas de point de défaillance unique
  - créer une branche (voir plus loin) ne coûte rien

Pourquoi git ?

- git est un système de contrôle de versions distribué :
  - chaque développeur a une copie complète de l'historique
  - pas de point de défaillance central
- Conçu pour :
  - la rapidité
  - les branches et les fusions
  - les gros projets
- git est :
  - le standard de fait (plus de 90 % de parts de marché)
  - utilisé par les grandes plateformes d'hébergement de code : GitHub, GitLab, Bitbucket

Origine de git

- git a été créé en 2005 par Linus Torvalds
- Motivation initiale :
  - gérer le code source du noyau Linux
  - des milliers de contributeurs
  - des exigences de performance très élevées
- Remplaçant du VCS propriétaire utilisé jusque-là (BitKeeper)
- Contraintes de conception :
  - rapidité
  - passage à l'échelle
  - développement massivement parallèle
  - intégrité de l'historique

Idées clefs de la conception de git

- git repose sur quelques principes forts :
  - un commit est identifié par un hachage cryptographique, pas par un numéro de version
  - l'historique est un graphe de commits
  - une branche est une simple référence dans ce graphe
- Conséquences pratiques :
  - créer une branche ne coûte rien
  - fusionner des commits est une opération fondamentale
  - l'intégrité de l'historique est vérifiable
- Beaucoup de concepts de git deviennent naturels une fois cette conception comprise

Dépôt et répertoire de travail

- Un dépôt git est un projet suivi par git
- Il contient :
  - vos fichiers (le répertoire de travail)
  - un sous-répertoire caché `.git/`
- Un répertoire n'est pas un dépôt git par défaut

Les trois zones

Zone	Contenu
Répertoire de travail	Vos fichiers actuels
Index (*staging area*)	Ce qui ira dans le prochain commit
Dépôt	L'historique des commits

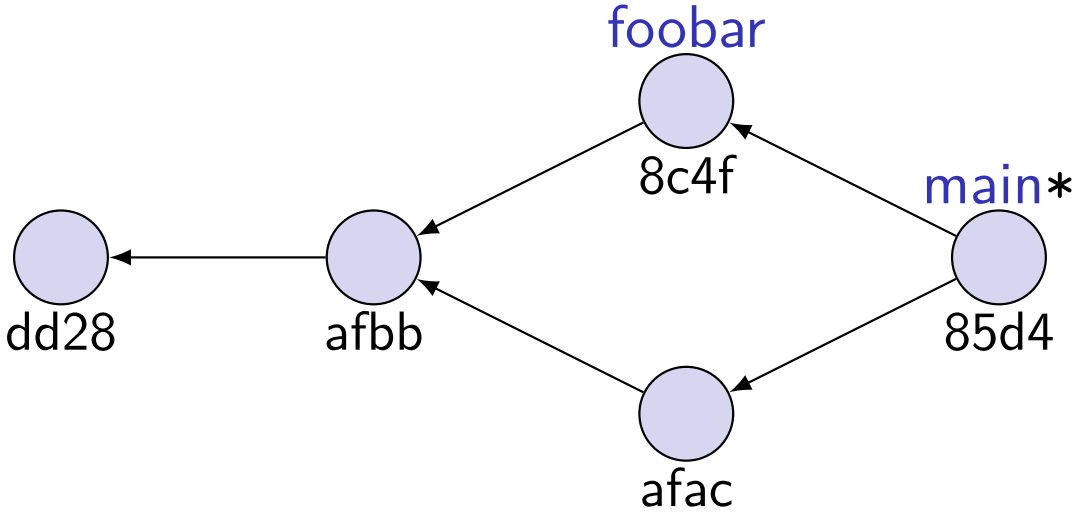
Commits

- Un commit est un instantané du projet (une version précise de tous ses fichiers)
- Chaque commit a :
  - un hachage unique qui sert d'identifiant : un nombre de 160 bits (algorithme SHA-1), écrit en hexadécimal, soit 40 chiffres hexadécimaux – on n'en utilise souvent que les premiers (p. ex., 7) quand il n'y a pas d'ambiguïté
  - un auteur (nom et adresse électronique) et une date précise
  - un message descriptif (décrivant cette version, ou plus souvent les changements par rapport à la ou aux versions précédentes)
- Les commits forment un graphe orienté acyclique (comme un arbre, sauf qu'un commit peut avoir plusieurs parents)

Branches et `HEAD`

- Une branche est un pointeur mobile vers un commit
- Il y a toujours une branche par défaut, appelée main ou master
- `HEAD` indique sur quel commit on se trouve
- Les branches permettent le travail en parallèle et l'expérimentation

Exemple : un dépôt et ses commits



- Chaque nœud est un commit (un instantané)
- Les commits sont identifiés par des hachages (simplifiés à 4 chiffres)
- Les flèches pointent vers le parent d'un commit
- Les branches (main, foobar) sont en bleu
- `HEAD` est indiqué par un *

Plan

Créer un dépôt

```
git init
```

- Crée un dépôt sans aucun commit
- Concrètement, crée un sous-répertoire `.git` qui contiendra tout l'historique
- Une fois pour toutes, indiquer à git qui vous êtes :

```
git config --global user.name "Jean Dupont"
git config --global user.email "jean@ens.psl.eu"
```

Connaître l'état du dépôt

```
git status
```

- `git status` répond à une question simple :
  « En quoi mon répertoire de travail diffère-t-il de `HEAD` ? »
- Il compare trois états :
  - le dernier commit (l'instantané courant)
  - l'index (ce qui ira dans le prochain commit)
  - le répertoire de travail (vos fichiers actuels)
- Informations typiques :
  - fichiers modifiés
  - fichiers nouveaux, non suivis (*untracked*)
  - modifications indexées, prêtes à être validées

`git add` : préparer le prochain commit

- `git add` sélectionne les modifications à inclure dans le prochain commit
- Il copie des modifications du répertoire de travail vers l'index

```
git add rapport.py
```

- Effet :
  - le contenu actuel de `rapport.py` est copié dans l'index
  - les modifications ultérieures de `rapport.py` ne sont pas indexées automatiquement
- Idée clef :
  - `git add` ne crée pas de commit
  - il définit *ce que le prochain commit contiendra*

`git rm` : enregistrer une suppression

- `git rm` supprime un fichier et indexe sa suppression
- L'historique reste intact

```
git rm vieilles_donnees.csv
```

- Effet :
  - le fichier est supprimé du répertoire de travail
  - la suppression est enregistrée dans l'index
- Après le prochain commit :
  - le fichier disparaît des instantanés suivants
  - il reste accessible dans les commits passés
- Conceptuellement : git suit les modifications des fichiers, suppressions comprises (`git mv` de même pour les renommages)

`git commit` : créer un instantané

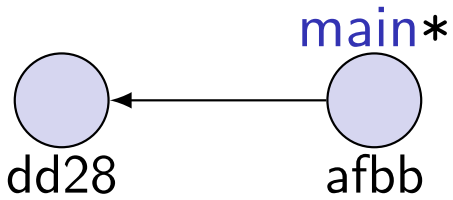
- `git commit` crée un nouveau commit à partir de l'index
- Le commit rejoint l'historique du dépôt

```
git commit -m "Corrige le tri"
```

- Effet :
  - un nouvel instantané est créé
  - il contient exactement ce qui était indexé
  - la branche courante pointe désormais vers ce commit
- Important :
  - seules les modifications indexées sont validées
  - les autres restent dans le répertoire de travail
  - sans `-m`, git ouvre votre éditeur pour le message

Exemple : deux commits

```
git init
git add Matrix.cpp Matrix.h
git commit -m "Classe Matrix"
git add main.cpp
git commit -m "Fonction main"
```



Consulter l'historique

- git stocke l'historique du projet sous forme d'un graphe de commits
- `git log` permet de parcourir cet historique

```
git log
```

- Affiche :
  - les commits, du plus récent au plus ancien
  - leurs identifiants (hachages)
  - auteur, date et message

```
git log --oneline --graph --all
```

- Vue compacte et visuelle :
  - une ligne par commit
  - graphe des branches et des fusions

Voir les différences

- git sait afficher les différences entre états
- `git diff` compare les fichiers ligne à ligne

```
git diff
```

- Affiche : les différences entre le répertoire de travail et l'index

```
git diff --staged
```

- Affiche : les différences entre l'index et le dernier commit

```
git diff afbb 85d4
```

- Affiche : les différences entre deux commits quelconques

Le fichier `.gitignore`

- Liste les fichiers que git ne doit pas suivre (fichiers compilés, fichiers produits par l'éditeur ou par LaTeX, etc.) – tout ce qui n'est pas strictement utile
- Ne concerne que les fichiers non suivis
- Les fichiers listés sont ignorés par `git status`, `git add`, etc.
- Le fichier `.gitignore` lui-même doit être ajouté à un commit !

Text

```
*.o
a.out
__pycache__/
*.aux
*.log
```

Restaurer des fichiers

```
git restore fichier.c
```

- Effet :
  - remplace `fichier.c` dans le répertoire de travail
  - par la version de l'index (c'est-à-dire la dernière version indexée ou validée)
- Usage : abandonner des modifications locales que l'on ne veut pas garder

```
git restore --staged fichier.c
```

- Effet :
  - retire `fichier.c` de l'index
  - sans toucher au répertoire de travail
- Ces opérations sont sûres : l'historique n'est pas modifié

Corriger et défaire des commits

- Parfois, c'est l'historique lui-même qu'il faut corriger

```
git commit --amend
```

- Effet : remplace le dernier commit par un nouveau construit à partir de l'index courant
- Usages : corriger un message, ajouter des modifications oubliées
- Important : le graphe change (nouveau hachage) ; à ne pas faire sur un commit déjà partagé

```
git revert <commit>
```

- Effet : crée un nouveau commit qui annule les modifications du commit donné ; l'historique est préservé, c'est la méthode à utiliser sur un commit partagé

Créer et changer de branche

- Une branche est un pointeur nommé vers un commit

```
git switch -c fonctionnalite-x
```

- Effet :
  - crée une branche nommée `fonctionnalite-x`
  - la fait pointer vers le commit courant
  - bascule le répertoire de travail sur cette branche

```
git switch main
```

- Effet :
  - bascule le répertoire de travail sur la branche `main`
  - met les fichiers dans l'état du commit de cette branche
- Idée clef : changer de branche change quel instantané on voit ; `git branch` liste les branches

Fusionner

- Fusionner (*merge*) combine les historiques de deux branches

```
git merge fonctionnalite-x
```

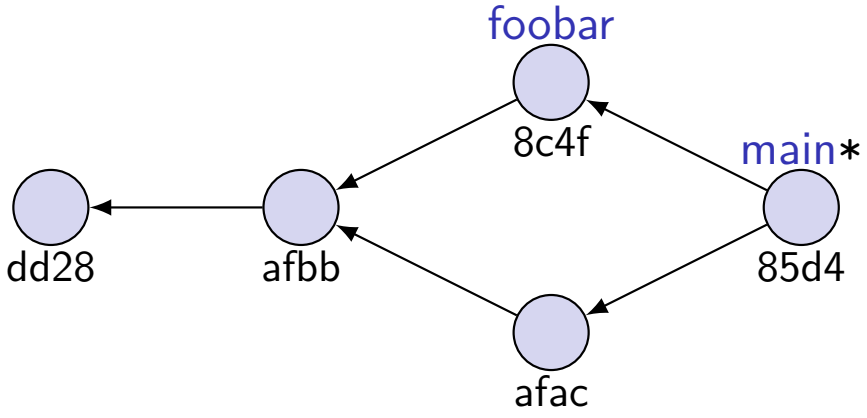
- Effet : intègre les modifications de `fonctionnalite-x` dans la branche courante
- Conceptuellement :
  - git trouve l'ancêtre commun
  - et combine les modifications faites depuis
- Résultat :
  - soit une avance rapide (*fast-forward*), si la branche courante n'avait pas bougé
  - soit un nouveau commit de fusion avec deux parents
- Important : la fusion préserve l'historique

Exemple : une branche fusionnée

```
git switch -c foobar
git add main.cpp
git commit -m "Améliore la fonction main"

git switch main
git add .gitignore
git commit -m "Ajoute un fichier .gitignore"

git merge foobar
```



Conflits de fusion

- Si les deux branches ont modifié les mêmes lignes, git ne peut pas décider : la fusion s'arrête sur un conflit
- Les fichiers concernés contiennent les deux versions :

Text

```
<<<<<<< HEAD
int n = 10;
=======
int n = 20;
>>>>>>> foobar
```

- Résolution :
  1. `git status` liste les fichiers en conflit
  2. éditer chaque fichier pour ne garder que la bonne version (les éditeurs modernes aident : VS Code, `git mergetool`)
  3. `git add fichier` pour marquer le conflit résolu
  4. `git commit` pour terminer la fusion
- `git merge --abort` pour renoncer et revenir à l'état d'avant

Démarrer un projet : `git clone`

- En pratique, on commence le plus souvent par cloner un dépôt existant

```
git clone https://github.com/pierre/projet.git
git clone git@github.com:pierre/projet.git
```

- Effet :
  - crée un nouveau répertoire local
  - y initialise un dépôt local
  - copie l'historique complet des commits
- Après le clonage :
  - le dépôt distant est connu sous le nom `origin`
  - un répertoire de travail est prêt à l'emploi
- Deux protocoles : HTTPS (jeton d'accès pour écrire) ou SSH, avec la clef configurée ce matin

Dépôts distants

- Un dépôt distant (*remote*) est un autre dépôt connu de git
- Il représente en général :
  - un serveur partagé (après y avoir cloné le dépôt)
  - ou la copie d'un autre développeur
- Les dépôts distants ont des noms : le plus souvent `origin` ; `git remote -v` les liste, `git remote add nom url` en ajoute un
- Conceptuellement :
  - un dépôt distant est simplement un autre graphe de commits
  - git garde trace de la relation entre votre historique et le sien (branches `origin/main`, etc.)

Récupérer et envoyer

- Collaborer, c'est échanger des commits

```
git pull
```

- Effet : récupère les commits du dépôt distant (`git fetch`) et les intègre à la branche courante (`git merge`)

```
git push
```

- Effet : envoie les commits locaux au dépôt distant, les rendant visibles aux autres ; `git push -u origin branche` la première fois pour une nouvelle branche
- Invariant : seuls des commits sont échangés, jamais des fichiers de travail
- Si quelqu'un a poussé entre-temps, `push` est refusé : faire d'abord `pull` (et résoudre les conflits éventuels)

Plan

Organisation du travail

- Une branche par fonctionnalité (ou par correction) : `main` reste toujours dans un état qui compile et passe les tests
- Des commits petits et cohérents, avec un message qui dit *pourquoi*, pas seulement quoi (la première ligne résume, le reste détaille)
- `pull` avant de commencer, `push` souvent : moins il y a d'écart, moins il y a de conflits
- Ne jamais valider ce qui se régénère (fichiers compilés, `.aux`, `.pdf`) ni de secrets (mots de passe, jetons) : `.gitignore`
- Ne pas réécrire (`-amend`, `rebase`) un historique déjà partagé

Plateformes d'hébergement

- GitHub, GitLab (instances à l'ENS, chez Inria), Codeberg… : hébergent des dépôts et ajoutent une couche collaborative : droits, suivi de problèmes (*issues*), revue de code, intégration continue
- Deux modèles pour contribuer :

  Dépôt partagé les membres du groupe ont les droits en écriture, chacun pousse ses branches sur le même dépôt

  Fork on copie le dépôt dans son propre espace (*fork*), on y pousse, puis on propose ses modifications au dépôt d'origine

- Dans le second cas, le dépôt d'origine devient un second dépôt distant, pour rester à jour :

```
git remote add upstream <url du dépôt d'origine>
git pull upstream main
```

Pull request et revue de code

- Une pull request (GitHub) ou merge request (GitLab) est une demande d'intégrer une branche dans une autre, typiquement dans `main` du dépôt d'origine
- Elle affiche le diff, permet de discuter ligne par ligne, de demander des modifications, et finit par une fusion (bouton *Merge*) par le mainteneur
- Tout commit poussé sur la branche s'ajoute automatiquement à la demande : on corrige en poussant, sans en créer une nouvelle
- Même seul, ou en petit groupe, la revue par un autre est le meilleur moyen de trouver les bugs et d'homogénéiser le code
- Cycle typique : `switch -c` → commits → `push` → pull request → revue → fusion → `pull` de `main`

Intégration continue

- Les plateformes peuvent exécuter des commandes à chaque push : compilation, tests, style, construction d'un document LaTeX, déploiement d'un site
- GitHub Actions : fichier YAML dans `.github/workflows/` ; GitLab CI : `.gitlab-ci.yml`

Yaml

```yaml
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: make
      - run: make test
```

- Le résultat apparaît sur chaque commit et chaque pull request : on ne fusionne que si les tests passent

git dans l'éditeur

- VS Code, emacs (magit), vim (fugitive) et les IDE intègrent git : mêmes concepts, interface graphique par-dessus
- Ligne de commande ou éditeur ?
  - Ligne de commande : universelle, précise, scriptable, fonctionne en SSH
  - Éditeur : confortable, excellents outils de diff et de résolution de conflits
  - Recommandation : comprendre la ligne de commande, utiliser l'éditeur comme aide

Pour aller plus loin

- `git stash` pour mettre de côté des modifications en cours ; `git rebase` pour rejouer une branche sur une autre et obtenir un historique linéaire ; `git bisect` pour trouver par dichotomie le commit qui a introduit un bug ; `git blame` pour savoir qui a écrit chaque ligne
- Apprendre le modèle mental, puis les commandes
- Pratiquer sur vos propres projets, et sur `https://learngitbranching.js.org/`
- *Pro Git*, S. Chacon et B. Straub, gratuit en ligne : `https://git-scm.com/book/`

Bibliography I


Arcep.
Baromètre 2025 de la transition vers IPv6 en France.
https://www.arcep.fr/cartes-et-donnees/nos-publications-chiffrees/transition-ipv6/, 2025.


IETF.
Request For Comments 791. Internet Protocol.
http://www.ietf.org/rfc/rfc0791.txt, September 1981.


IETF.
Request For Comments 1034. Domain names—concepts and facilities.
http://www.ietf.org/rfc/rfc1034.txt, June 1999.

Bibliography II


IETF.
Request For Comments 3986. Uniform Resource Identifier (URI) : Generic syntax.
https://www.rfc-editor.org/rfc/rfc3986, January 2005.


IETF.
Request For Comments 8200. Internet Protocol, version 6 (IPv6) specification.
https://www.rfc-editor.org/rfc/rfc8200, July 2017.


IETF.
Request For Comments 9110. HTTP semantics.
https://www.rfc-editor.org/rfc/rfc9110, June 2022a.

Bibliography III


IETF.
Request For Comments 9293. Transmission Control Protocol (TCP).
https://www.rfc-editor.org/rfc/rfc9293, August 2022b.