TELECOM SudParis

Introduction à GIT pour TSP1A

Daniel Ranc

RS2M

daniel.ranc@telecom-sudparis.eu

TELECOM SudParis

Plan

- Motivations et principes généraux
- Guide de bonne pratique
- Utilisation de GitLab TSP

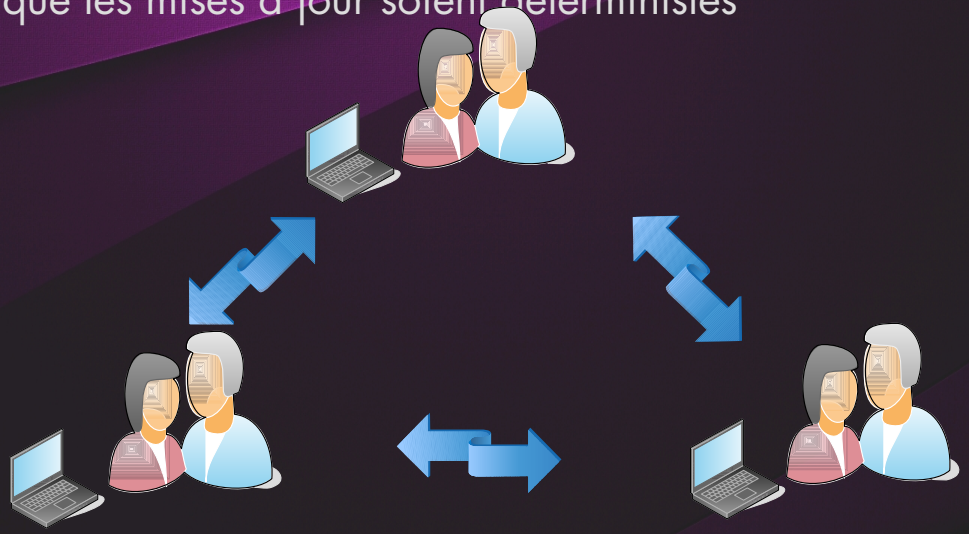
TELECOM SudParis

Plan

- **Motivations et principes généraux**
- **Guide de bonne pratique**
- **Utilisation de GitLab TSP**

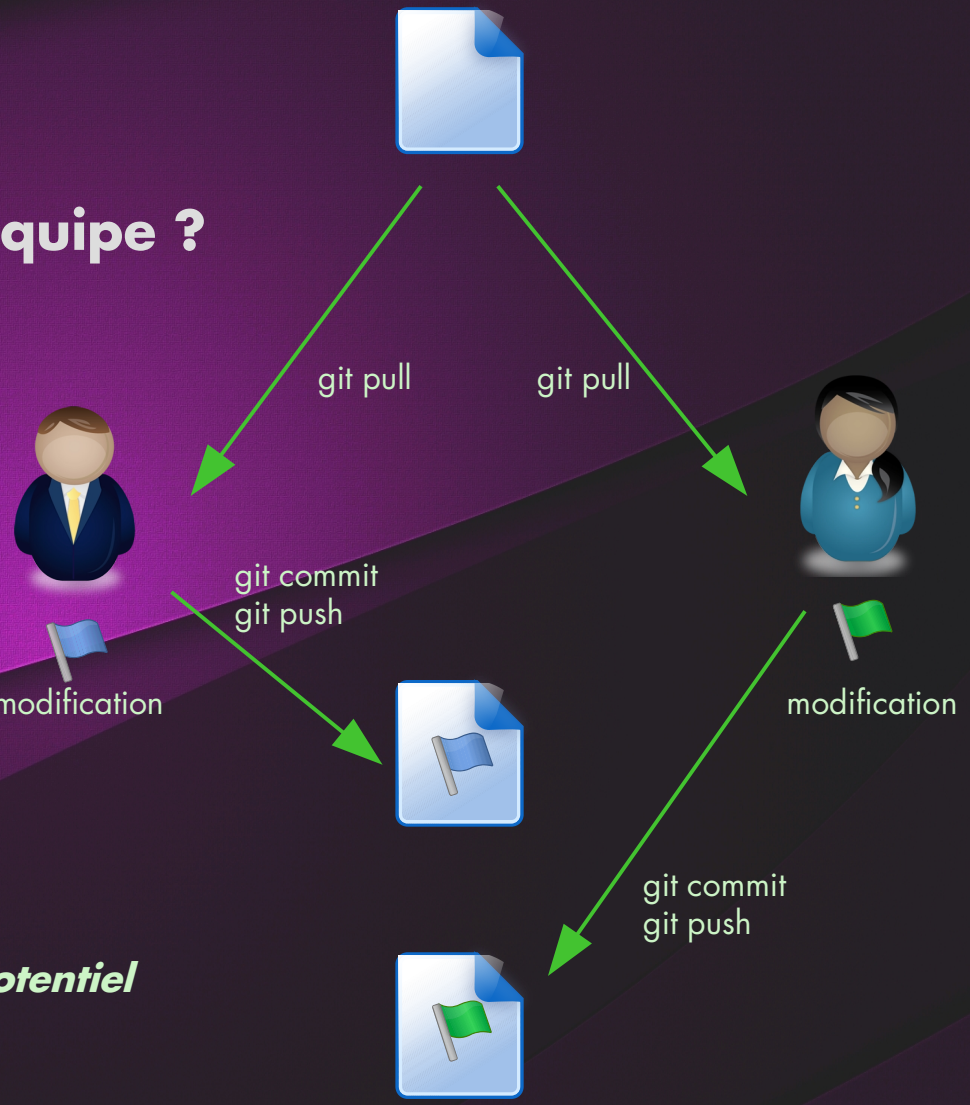
Motivations

- **Comment développer en équipe ?**
  - **Besoin d'échanger les fichiers de code**
    - À chaque modification des interfaces externes du code
    - À chaque ajout d'une propriété
  - **Besoin de contrôle des versions**
    - Pour que les mises à jour soient déterministes



Motivations

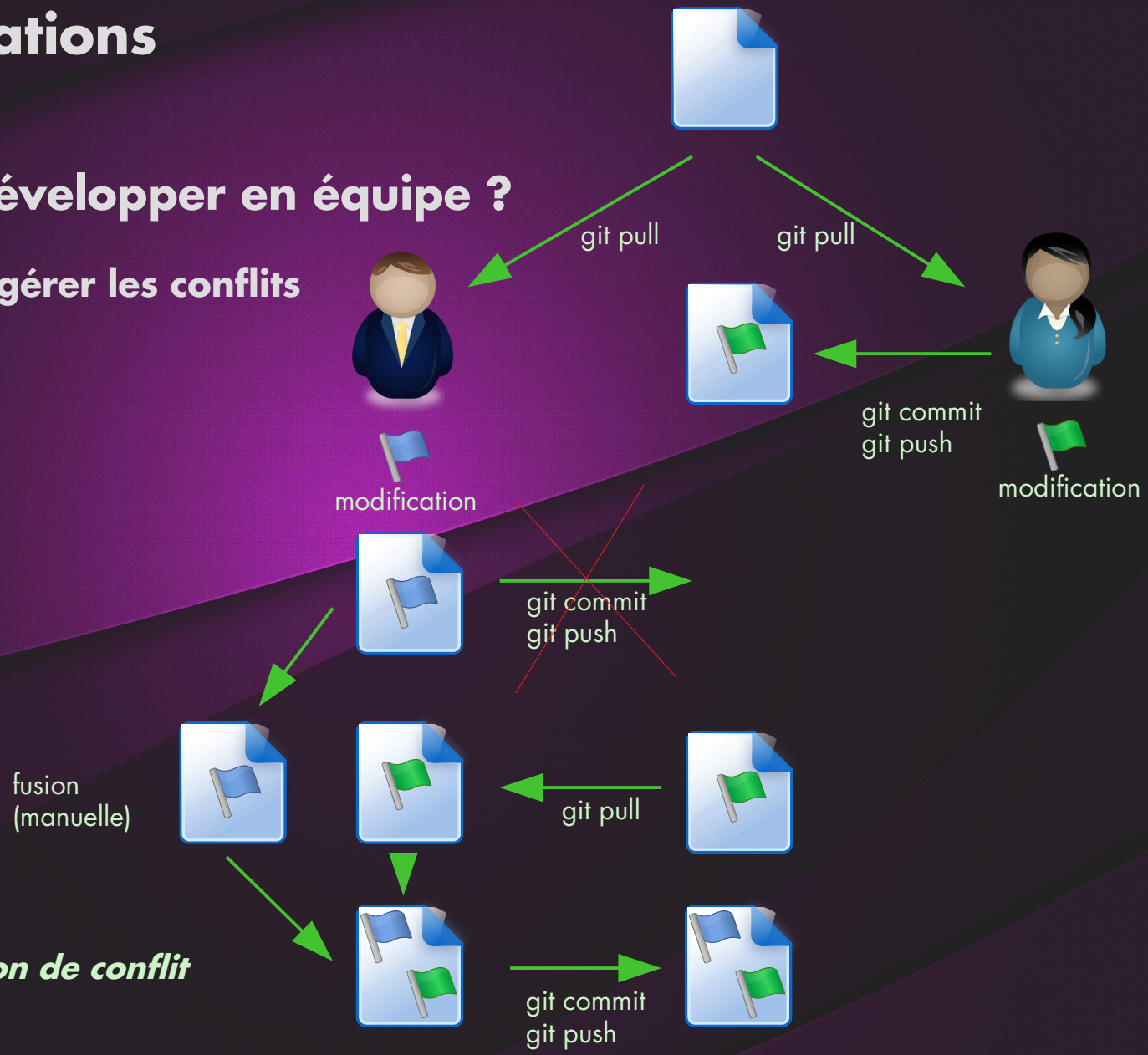
- Comment développer en équipe ?
  - Besoin de gérer les conflits



Exemple de conflit potentiel

Motivations

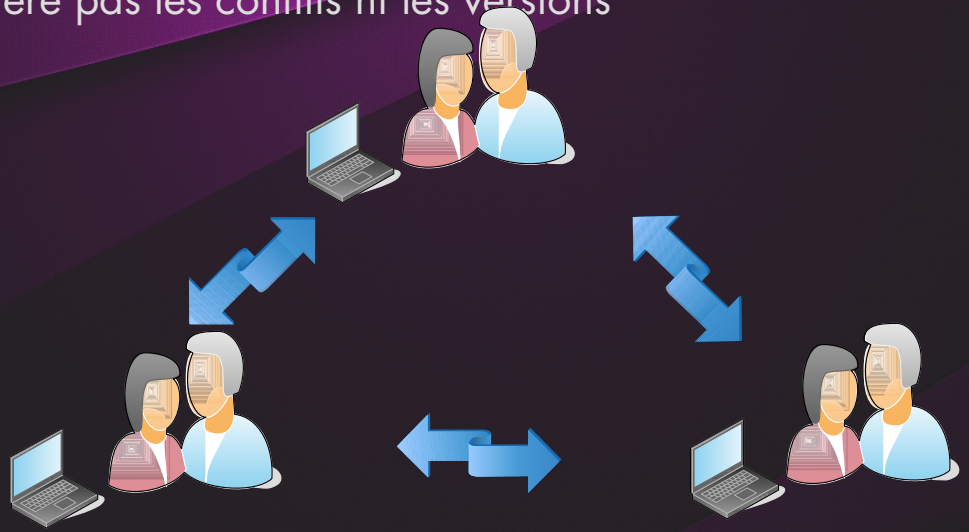
- Comment développer en équipe ?
  - Besoin de gérer les conflits



Exemple de résolution de conflit

Motivations

- **Comment développer en équipe ?**
  - **Serveur central type Google Docs**
    - Ne gère pas les conflits ni les versions
  - **Échanges par duplication (email, dropbox, ...)**
    - Risque d'incohérence entre les copies locales
    - Ne gère pas les conflits ni les versions



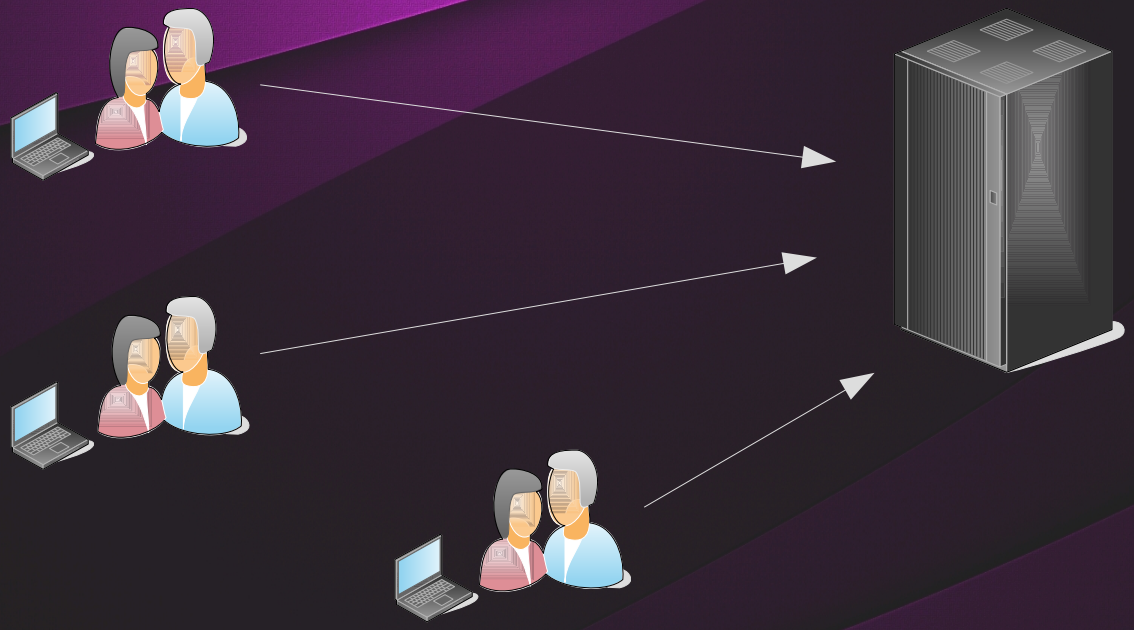
TELECOM SudParis

Motivations

- **Comment développer en équipe ?**
  - **Dès les années 80 des *systèmes de gestion de version* sont utilisés**
    - Historique : RCS, CVS, SVN
    - Le standard actuel est GIT
      - Écrit en 2005 par Linus Torvalds pour faciliter le développement réparti du noyau Linux (des centaines de contributeurs dans le monde entier)
      - Complètement distribué
    - Usage au travers de *forges logicielles*
      - Environnement centralisé complet composé d'un système de gestion de versions et d'outils :
        * Documentation
        * Suivi de gestion de bugs
      - Exemples : SourceForge, GitHub, GitLab

Motivations

- **Comment développer en équipe ?**
  - **Serveur central "forge logicielle" assurant**
    - Accès sécurisé
    - Travail en groupe
    - Moyens de gestion de la cohérence et des conflits



TELECOM SudParis

Vocabulaire

- **Dépôt**
  - **Espace disque du serveur contenant toutes les informations relatives à l'historique des versions (répertoires, fichiers, dates et auteurs des modifications, etc.)**
    - Sous git, le dépôt central s'appelle par convention *origin*.

TELECOM SudParis

Vocabulaire

- **Dépôt**
  - **Espace disque du serveur contenant toutes les informations relatives à l'historique des versions (répertoires, fichiers, dates et auteurs des modifications, etc.)**
    - Sous git, le dépôt central s'appelle par convention *origin*.
- **Copie de travail**
  - **Copie locale à la machine de l'utilisateur susceptible d'être modifiée par l'utilisateur, et aussi d'être synchronisée avec le dépôt.**

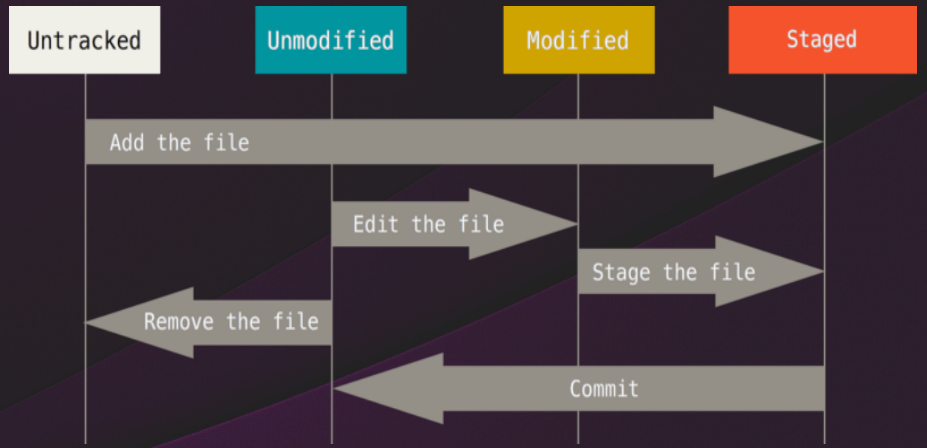
TELECOM SudParis

Vocabulaire

- **Dépôt**
  - **Espace disque du serveur contenant toutes les informations relatives à l'historique des versions (répertoires, fichiers, dates et auteurs des modifications, etc.)**
    - Sous git, le dépôt central s'appelle par convention *origin*.
- **Copie de travail**
  - **Copie locale à la machine de l'utilisateur susceptible d'être modifiée par l'utilisateur, et aussi d'être synchronisée avec le dépôt.**
- **Instantané/snapshot**
  - **État déterminé des fichiers étiqueté par un nombre et un message. Le système de gestion de versions conserve l'historique des instantanés et permet éventuellement de les reconstituer a posteriori.**

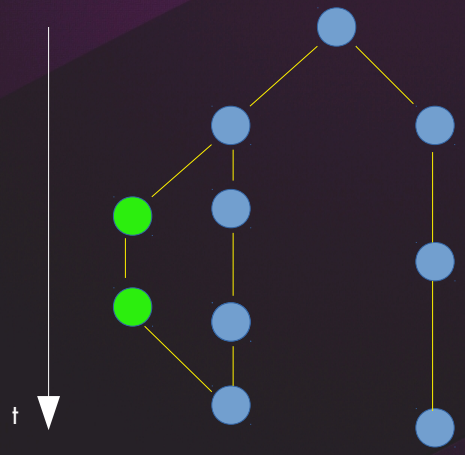
Concepts – zone de transit/staging area

- **Sous git le chemin des fichiers de travail vers le dépôt central suit une procédure en trois temps**
  - **1er temps : attachement du ou des fichier(s) à la zone de transit**
    - `git add <monfichier>`
  - **2ème temps : le commit**
    - `git commit -m "message explicatif"`
    - cette opération créée un état/un instantané
  - **3ème temps : l'instantané est poussé sur le dépôt**
    - `git push`



Concepts – graphe des révisions

- L'historique des états du projet (sanctionnés par des *commits*) constitue un graphe orienté
- La bonne pratique consiste à gérer plusieurs *branches* dans ce graphe
  - *permanentes*, axes de développement principaux
  - *temporaires*, consacrées au développement de modules avant leur intégration ; le destin attendu d'une branche temporaire est de fusionner avec une branche permanente



TELECOM SudParis

Principales opérations git

- **git init**
  - création d'une arborescence de travail
- **git clone <url dépôt distant>**
  - réalise une copie du dépôt distant, immédiatement sous contrôle de versions
- **git status**
  - informe sur l'état actuel du projet
- **git checkout -b <mabranche>**
  - création d'une nouvelle branche
- **git checkout <mabranche>**
  - commute vers la branche indiquée

TELECOM SudParis

Principales opérations git

- **git add <monfichier>**
  - ajoute le fichier à la zone de transit
- **git commit -m "message explicatif"**
  - regroupe tous les fichiers de la zone de transit en une seule transaction, qui représentera un instantané
- **git push**
  - pousse les fichiers depuis la zone de transit vers *origin*
- **git fetch**
  - tire l'état le plus récent depuis *origin* ; doit être suivi d'un *merge* pour être effectif
- **git merge <mabranche>**
  - fusionne la branche indiquée avec la branche courante, avec détection de conflits éventuels
- **git pull (= fetch + merge)**

TELECOM SudParis

Quelques observations

- D'autres opérations sont possibles
  - les opérations listées précédemment sont suffisantes !
- La plupart des opérations sont locales
  - seuls push, fetch et pull interagissent avec *origin*

TELECOM SudParis

Plan

- Motivations et principes généraux
- Guide de bonne pratique
- Utilisation de GitLab TSP

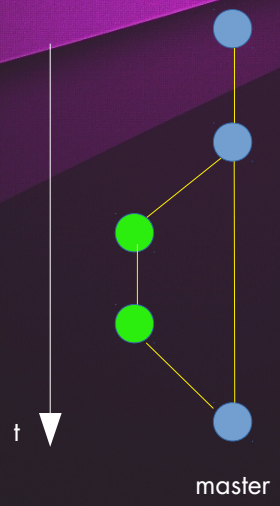
TELECOM SudParis

Guide de bonne pratique

- **Le modèle de git est ouvert**
  - **l'utilisateur est entièrement libre de s'organiser comme il l'entend**
  - **cette liberté comporte des risques dont il faut être conscient :**
    - conventions et usages différents d'une équipe à l'autre
    - usage menant à une complexité devenant ingérable
    - opérations complexes menant à des erreurs, toujours difficiles (et parfois **très** difficiles) à récupérer
- **La sagesse impose d'adopter des conventions et des pratiques éprouvées**
  - **"A successful Git branching model", Vincent Driessen, 2010**
  - **En cours/TP : *procédure simplifiée***

Guide de bonne pratique

- **Détail de la *procédure simplifiée***
  - **Règle 1: une branche permanente, `master`**
  - **Règle 2 : les contributions se font sur des branches *topiques* et sont fusionnées avec `master` <u>après validation</u>**



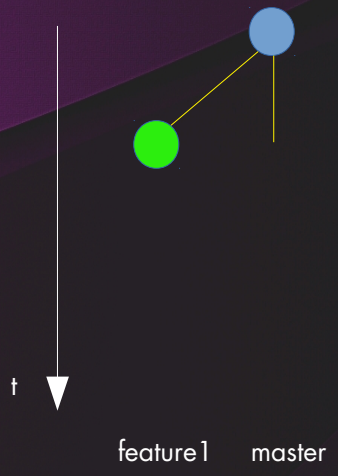
TELECOM SudParis

Guide de bonne pratique

- **Scénario-type**
  - **Initialisation d'un répertoire de travail**
    - `$ git init`
      - nota bene : pour un projet sous GitLab il faut d'abord créer le projet avec l'interface web, puis effectuer un clone.
  - **Commit initial**
    - `$ git commit –allow-empty -m 'commit initial'`
      - L'option –allow-empty permet d'économiser la création d'un fichier (README p.ex.)
  - **Création de la branche topique**
      - (master existe déjà)
    - `$ git checkout -b feature1`
      - (ceci commute sur la branche feature1)

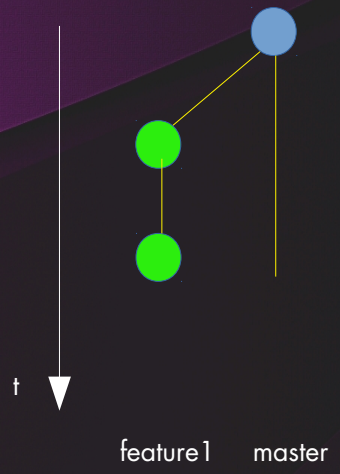
Guide de bonne pratique

- **Scénario-type**
  - **Création/édition d'un fichier**
    - `$ nano monfichier.txt`
  - **Observation de l'état actuel**
    - `$ git status`
      - indique que monfichier.txt n'est pas pris en compte
  - **Ajout du fichier à la zone de transit**
    - `$ git add monfichier.txt`
  - **Commit**
    - `$ git commit -m 'création de monfichier.txt'`



Guide de bonne pratique

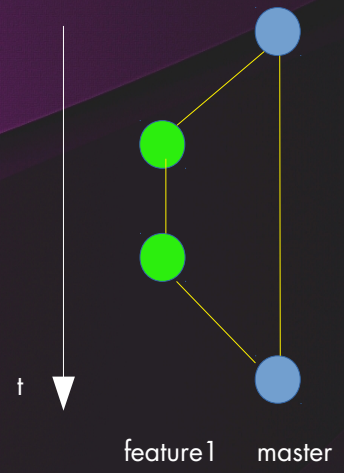
- **Scénario-type**
  - **Modification du fichier**
    - `$ nano monfichier.txt`
    - un `git status` indique que le fichier est modifié
  - **Ajout à la zone de transit**
    - `$ git add monfichier.txt`
  - **Second commit suite à modification**
    - `$ git commit -m 'modification de monfichier.txt'`



Guide de bonne pratique

- **Scénario-type**
  - **Fusion de la branche temporaire avec `master`,**
  - **après validation de ce code**
    - `$ git checkout master # commutation`

      `sur la branche destinataire`
    - `$ git merge --no-ff feature1  # fusion`
      - **l'option –no-ff préserve l'historique de `feature1`**
  - **Ce nouvel état du projet doit être poussé sur le serveur**
    - permettant l'accès à ce nouveau code par les autres membres de l'équipe
    - `$ git push origin master`



TELECOM SudParis

Bouées de sauvetage

- **En cas de catastrophe :**
  - **D'abord, repérer les étiquettes des commits**
    - `$ git log --graph`
  - **Défaire un commit publié sur la forge**
    - `$ git revert <hash>`
      - Ajoute un nouveau commit défaisant celui indiqué ; faire un `push` ensuite
  - **Revenir à une révision locale**
    - `$ git reset --hard <hash>`
      - Retourne au commit indiqué en supprimant (localement) tout l'historique depuis celui-ci

Ref : https://github.com/blog/2019-how-to-undo-almost-anything-with-git

TELECOM SudParis

Plan

- Motivations et principes généraux
- Guide de bonne pratique
- Utilisation de GitLab TSP

Utilisation de GitLab TSP

- **Forge logicielle Git**
  - **Dépôt centralisé, sécurisé avec interface d'administration web**
    - concurrent de GitHub, BitBucket etc.
  - **Accès par CAS/Shibboleth**
  - **Usage authentifié par clé publique SSH**
  - **Gère les équipes-projet**
  - `https://gitlab.tem-tsp.eu`

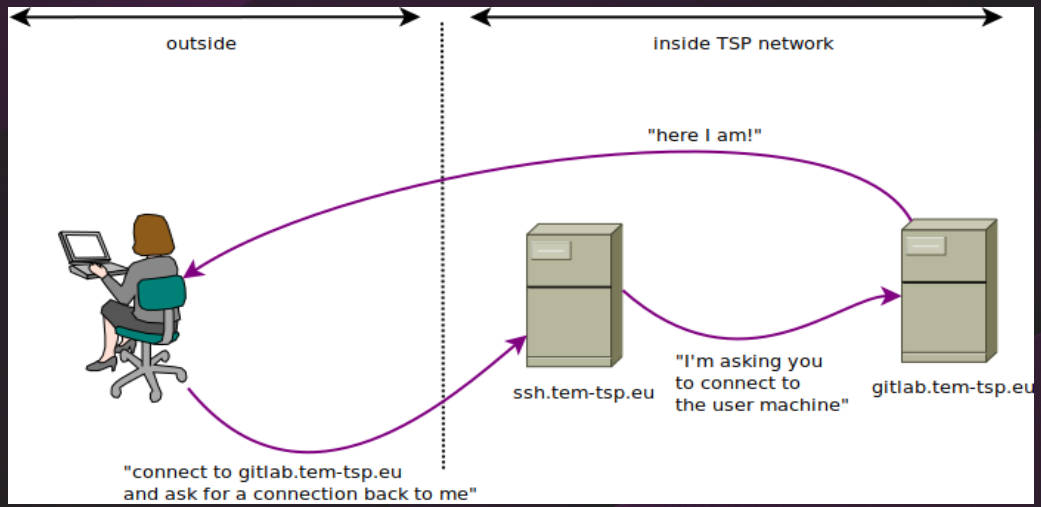
GitLab

Utilisation de GitLab TSP

- **Principes de l'authentification**
  - **L'utilisateur associe la *clé publique SSH* de son compte Linux à son compte GitLab**
    - par l'interface web
  - **L'accès à GitLab par Git depuis le terminal est en SSH**
  - **Cette procédure permet d'authentifier l'utilisateur**
  - **Plusieurs clés publiques peuvent être associées au compte GitLab, permettant l'accès depuis un compte Linux personnel**

Utilisation de GitLab TSP

- **Gestion de l'itinérance**
  - **Le serveur GitLab n'est pas accessible, normalement, depuis l'extérieur de TSP**
    - ni la page web, ni le service Git
    - prudence de la politique de gestion DISI
  - **Une procédure passant par `ssh.tem-tsp.eu` et mettant en jeu un rebond permet de contourner de façon sécurisée cette limitation**
    - cf. page Moodle du module



TELECOM SudParis

Conclusion

- L'ensemble de ces concepts sera maintenant vu en TP :
  - Mise en place d'un projet versionné sous GitLab
  - Pratique d'une procédure Git simplifiée
  - Gestion d'un conflit
- Un second TP aborde la procédure Git-flow
  - En auto-formation

TELECOM SudParis

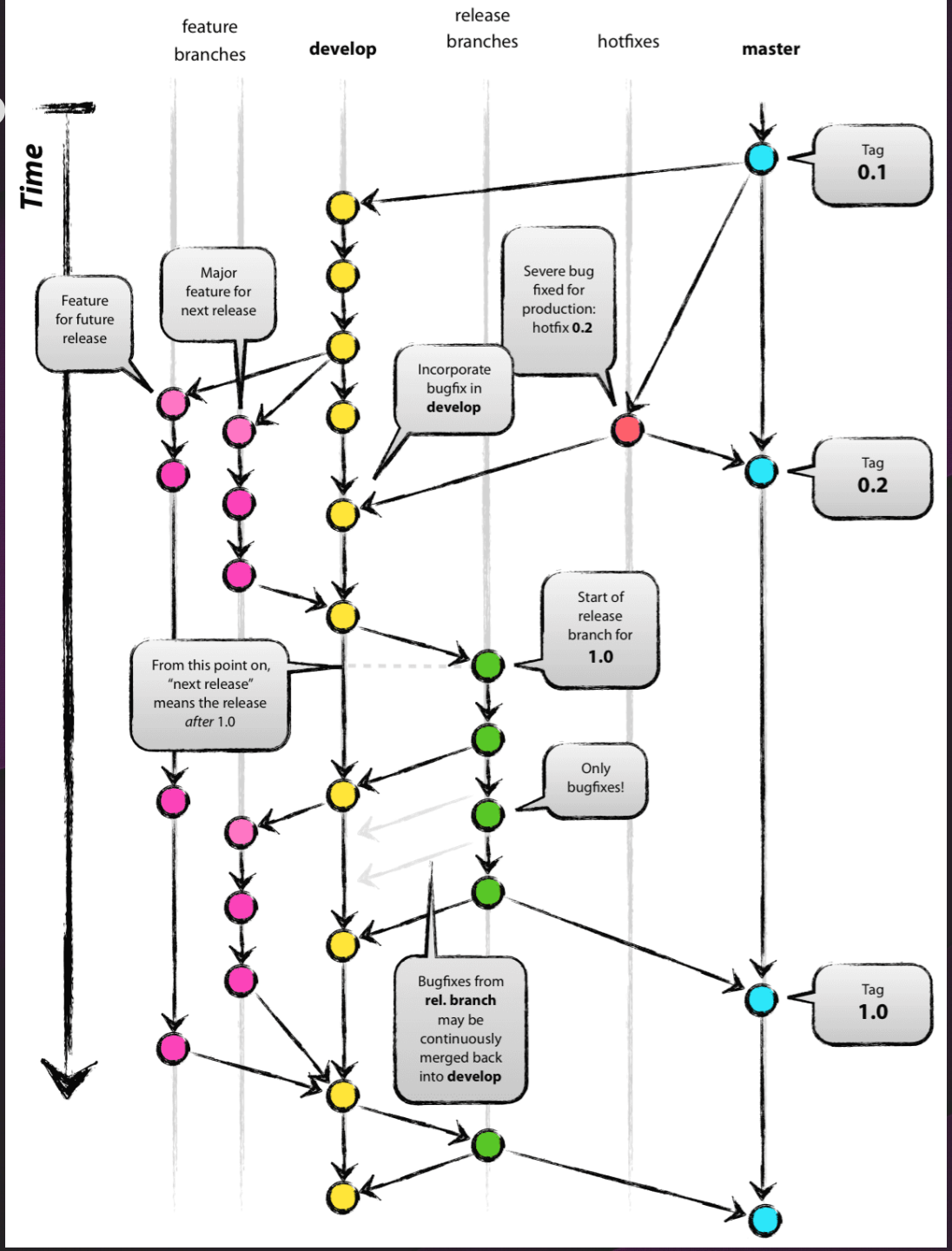
Any questions?

TELECOM SudParis

Supplément : Git-flow

TELECOM SudParis

Sup



TELECOM SudParis

Supplément : Git-flow

- branches *master* et *develop*
  - master : la branche de référence supportant en permanence des versions publiables/déployables (*releases*)
  - develop : la branche de développement sur laquelle se greffent les branches de modules (*features*)
  - les releases s'obtiennent soit par une fusion directe de develop sur master, soit par une branche intermédiaire permettant la correction de bugs, puis fusion sur master
- branches de modules (features)
  - création au fur et à mesure des besoins
  - fusion avec develop dès que le module est validé

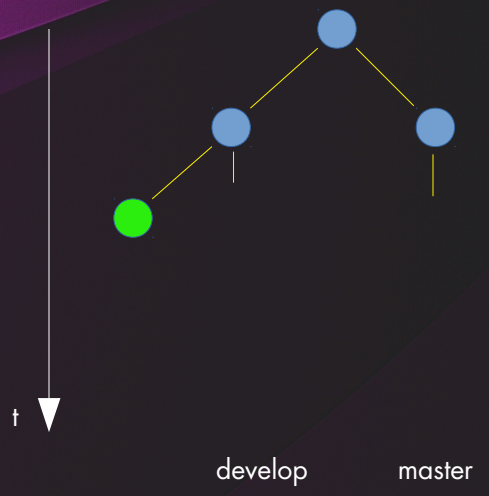
TELECOM SudParis

Supplément : Git-flow

- **Scénario-type**
  - **initialisation d'un répertoire de travail**
    - `$ git init`
      - nota bene : pour un projet sous GitLab il faut d'abord créer le projet avec l'interface web, puis effectuer un `clone`.
  - **commit initial**
    - `$ git commit --allow-empty -m 'commit initial'`
      - L'option –allow-empty permet d'économiser la création d'un fichier (README p.ex.)
  - **création de la branche develop**
      - (master existe déjà)
    - `$ git checkout -b develop`

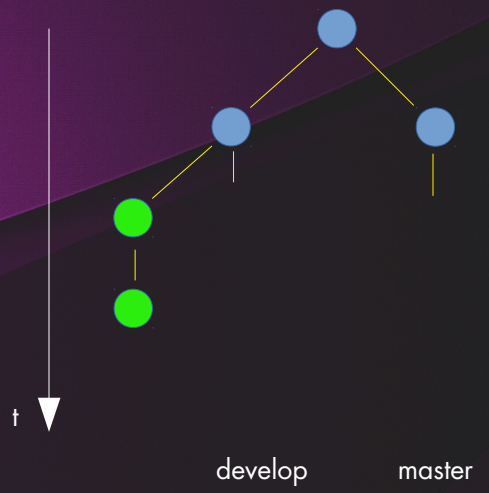
Supplément : Git-flow

- **Scénario-type**
  - **commutation sur develop et création d'une branche**
    - `$ git checkout develop`
    - `$ git checkout -b mabranche`
  - **création/édition d'un fichier**
    - `$ nano monfichier.txt`
  - **observation de l'état actuel**
    - `$ git status`
      - indique que monfichier.txt n'est pas pris en compte
  - **ajout du fichier à la zone de transit**
    - `$ git add monfichier.txt`
  - **commit**
    - `$ git commit -m 'création de monfichier.txt'`



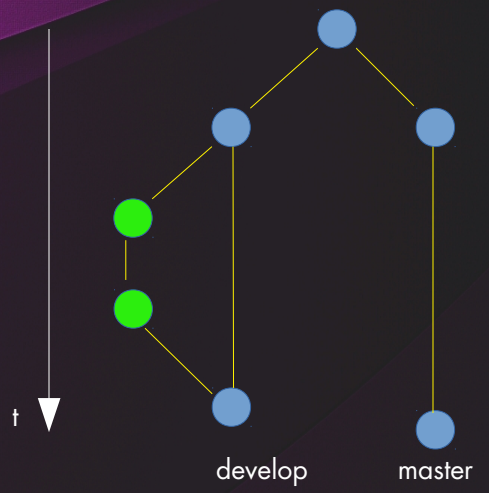
Supplément : Git-flow

- **Scénario-type**
  - **modification du fichier**
    - `$ nano monfichier.txt`
    - un git status indique que le fichier est modifié
  - **ajout à la zone de transit**
    - `$ git add monfichier.txt`
  - **second commit suite à modification**
    - `$ git commit -m 'modification du fichier'`



Supplément : Git-flow

- **Scénario-type**
  - **fusion de la branche temporaire avec develop**
    - `$ git checkout develop` # commutation sur la branche destinataire
    - `$ git merge --no-ff mabranche` # fusion
    - l'option –no-ff préserve l'historique de mabranche



Supplément : Git-flow

- **Scénario-type**
  - **fusion de la branche develop avec master**
    - `$ git checkout master` # commutation sur la branche destinataire
    - `$ git merge --no-ff develop` # fusion
    - l'option –no-ff préserve l'historique de develop