

# **M04 Systèmes & Réseaux**

Licence Professionnelle

D.Buscaldi

# Programme

- Aide au développement de logiciels:
  - Apache ANT
- Introduction aux systèmes distribués
  - Objets distribués: Java RMI
- Systèmes de gestion de version:
  - SGV client-serveur: Subversion (SVN)
  - SGV distribué: GIT
- Développement sur les Clouds
  - Amazon Web Services

# Modalité

- Séances cours/TD et TPs en salle machine
  - Groupes de 3 étudiants
  - Linux
- Note:
  - 1/2 Contrôle final (questions, exercices)
  - 1/2 Évaluation des TPs / projet

# 1. Les moteurs de production

## Ant

# Les moteurs de production

- En anglais: **Build Automation Tools (BAT)**
- Logiciels dont la fonction principale consiste à automatiser l'ensemble des actions (préprocessing, compilation, etc.) contribuant, à partir de données sources, à la production d'un logiciel opérationnel
- Fonctions:
  - Gestion des dépendances
  - Génération de documentation
  - Maintenance du code

# Les moteurs de production

- Historiquement le premier BAT a été GNU Make
- Aujourd'hui, on en trouve plusieurs
  - Ant
  - Gradle
  - Maven
  - Jenkins
  - ...
- Désormais souvent intégrés dans les IDEs (Visual Studio, Eclipse...)

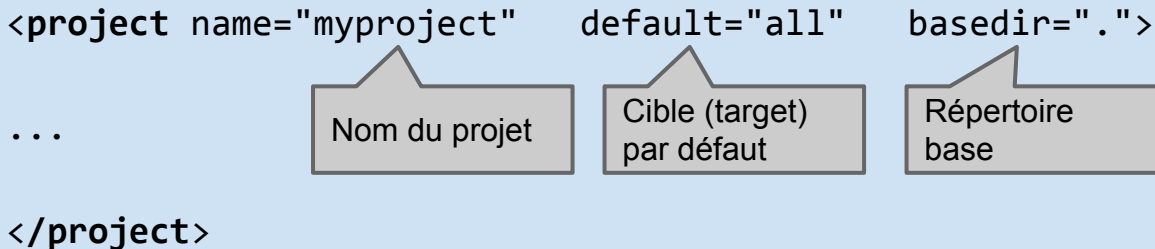
# Apache ANT



- ANT (Another Neat Tool) est un outil de construction (build) indépendant de toute plateforme
- ANT permet l'intégration continue et améliore la collaboration dans des projets complexes
- Langage cible: Java
- <http://ant.apache.org/index.html>

# Projets, cibles et tâches

- Les scripts ANT (buildfiles ou fichiers *build*) sont écrit en XML
  - Nom par défaut: **build.xml**
- Chaque buildfile correspond à un seul projet
  - Un buildfile commence et termine avec un tag **project**:



```
<project name="myproject" default="all" basedir=".">  
...  
</project>
```

The diagram shows an XML tag `<project>` with three attributes: `name="myproject"`, `default="all"`, and `basedir="."`. Below each attribute is a callout box with a pointer to the attribute value:

- `name="myproject"` points to "Nom du projet"
- `default="all"` points to "Cible (target) par défaut"
- `basedir="."` points to "Répertoire base"

Ellipses (`...`) are shown between the opening and closing tags.



# Projets, cibles et tâches

- Chaque projet contient une ou plusieurs cibles (**target**)
  - Chaque cible contient un ensemble de tâches (task) à exécuter
  - Tâches courantes: copier des fichiers, créer des répertoires, compiler, créer des archives...

```
<project name="myproject"    default="all"    basedir=".">
```

```
    <target name="all" depends="clean,build,test,docs,deploy">...</target>
```

Nom de la cible

Cibles requises

Tâches

```
    <target name="clean">...</target>
```

```
    <target name="build" depends="clean">...</target>
```

```
    <target name="deploy" depends="build, test"> ...</target>
```

```
    ...
```

```
</project>
```

# Projets, cibles et tâches

- Si une cible nécessite une autre cible (attribute *depends*), elle ne peut pas être complétée si la cible requise n'est pas terminée

```
<project name="myproject" default="all"
```

Pour *all* il faut terminer d'abord *clean*,  
*build*, *test*, *docs* et *deploy*

```
<target name="all" depends="clean,build,test,docs,deploy">...</target>
```

Nom de la cible

Cibles requises

Tâches

```
<target name="clean">...</target>
```

```
<target name="build" depends="clean">...</target>
```

```
<target name="deploy" depends="build, test"> ...</target>
```

```
...
```

```
</project>
```

# Tâches

- Dans chaque cible on a une liste de **tâches** ou **actions**:
- Les actions disponibles sont nombreuses mais limitées: on peut compiler, effacer/créer un dossier, créer des archives, demander de l'input, etc...

- Exemples:

```
<delete dir="classes"/> <!-- effacer le dossier "classes" -->
```

```
<mkdir dir="classes"/> <!-- créer un dossier nommé "classes" -->
```

```
<javac srcdir="." destdir="classes"/> <!-- compiler -->
```

```
<input message="Continuer (o/n) ?" validargs="o,n" ... />
```

```
<zip destfile="archive.zip" basedir="docs/manual" />
```

...

- La liste complète est disponible sur la page de ANT:
  - <http://ant.apache.org/manual/tasksoverview.html>

# Graphe de dépendances

- Graphe orienté  $G=\langle E, V \rangle$  qui montre les dépendances des cibles
  - $V$ = ensemble des cibles;
  - $E$ =ensemble des relations de dépendance;
    - $e(v_1, v_2) \in E \Leftrightarrow v_2$  dépends de  $v_1$
  - Exemple:

# Graphe de dépendances

- Exemple:

```
<project name="myproject"    default="all"    basedir=".">
  <target name="all" depends="clean,build,deploy">...</target>
  <target name="clean">...</target>
  <target name="build" depends="clean">...</target>
  <target name="deploy" depends="build, test"> ...</target>
  <target name="test"> ...</target>
</project>
```

## Cibles:

- all
- clean
- build
- deploy
- test

V= {all, clean, build, deploy, test}

## Dépendances:

- all ← clean, build, deploy
- build ← clean
- deploy ← build, test

E= {(clean, all), (build, all), (deploy, all), (clean, build), (build, deploy), (test, deploy)}

# Graphe de dépendances

## Cibles:

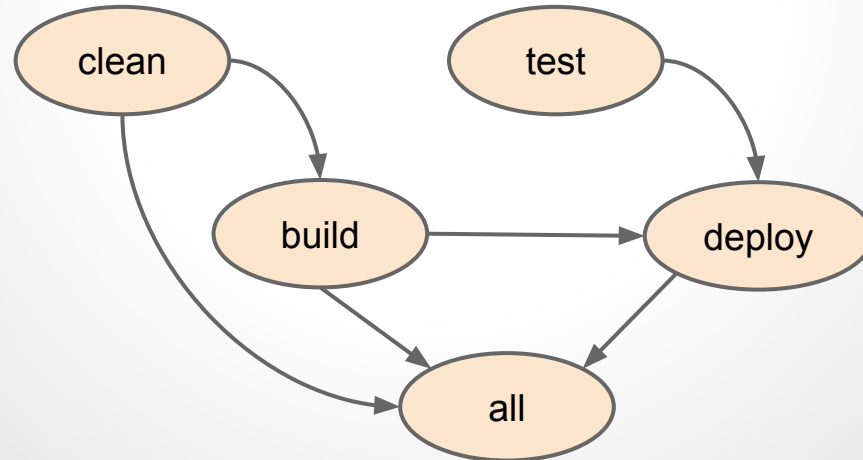
- all
- clean
- build
- deploy
- test

$V = \{\text{all, clean, build, deploy, test}\}$

## Dépendances:

- $\text{all} \leftarrow \text{clean, build, deploy}$
- $\text{build} \leftarrow \text{clean}$
- $\text{deploy} \leftarrow \text{build, test}$

$E = \{(\text{clean, all}), (\text{build, all}), (\text{deploy, all}), (\text{clean, build}), (\text{build, deploy}), (\text{test, deploy})\}$



# Example

## **HelloWorld.java**

```
package xptoolkit;  
public class HelloWorld{  
    public static void main(String []args){  
        System.out.println("Hello World!");  
    }  
}
```

## **build.xml**

```
<project name="hello" default="compile">  
    <target name="prepare">  
        <mkdir dir="/tmp/classes"/>  
    </target>  
    <target name="compile" depends="prepare">  
        <javac srcdir="./src" destdir="/tmp/classes"/>  
    </target>  
</project>
```

# Propriétés

- Les propriétés servent à définir des variables utilisées souvent
- Pour définir une propriété:
  - **<property name="nom" value="valeur" />**
  - Exemple: `<property name="outputdir" value="/tmp" />`
- Pour utiliser une propriété précédemment définie: **`${nom}`**
  - Exemple:

```
<project name="hello" default="compile">
  <property name="outputdir" value="/tmp"/>
  <target name="prepare">
    <mkdir dir="${outputdir}/classes"/>
  </target>
  <target name="compile" depends="prepare">
    <javac srcdir="./src" destdir="${outputdir}/classes"/>
  </target>
</project>
```



# Propriétés

- On peut définir des propriétés dans un fichier séparé et le charger
  - Exemple:

*Build.xml:*

```
<project name="hello" default="compile">  
  <property file="build.properties"/>  
  <target name="prepare">  
    <mkdir dir="${outputdir}/classes"/>  
  </target>  
  <target name="compile" depends="prepare">  
    <javac srcdir="./src" destdir="${outputdir}/classes"/>  
  </target>  
</project>
```

*build.properties:*

outputdir = /tmp

buildversion = 3.3.2

# Conditions (if / unless)

- On peut introduire des conditions pour l'exécution de certaines cibles
  - `<target name="setupProduction" if="production"> ... </target>`
  - `<target name="setupDevelopment" unless="production"> ... </target>`
- Exemple:

```
$ ant -buildfile build.xml -Dproduction=true
```

- ANT lancera la cible “setupProduction”

# Listes de fichiers

- On peut définir des listes de fichiers en fonction de motifs (*patterns*) avec **fileset**:

```
<fileset dir="${src}" casesensitive="yes">  
  <include name="**/*.java"/>  
  <exclude name="**/*Stub*"/>  
</fileset>
```

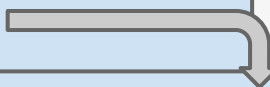
- Ou une liste explicite avec **filelist**:

```
<filelist id="config.files" dir="${webapp.src.folder}">  
  <file name="applicationConfig.xml"/>  
  <file name="faces-config.xml"/>  
  <file name="web.xml"/>  
  <file name="portlet.xml"/>  
</filelist>
```

# Exécution d'un programme

- On peut utiliser la tâche **java** pour lancer un programme:

```
<target name="Test" >  
  <java classname="TestClass" >  
    <classpath refid="projet.classpath"/>  
  </java>  
</target>
```



```
<!-- Definition du classpath du projet -->  
<path id="projet.classpath">  
  <fileset dir="${projet.lib.dir}">  
    <include name="*.jar"/>  
  </fileset>  
  <pathelement location="${projet.bin.dir}" />  
</path>
```

# Interaction avec l'utilisateur

- La tâche **input** permet de interagir avec l'utilisateur:

```
<input  
  message="Please enter db-username:"  
  addproperty="db.user"  
>
```

La valeur entrée par l'utilisateur sera stockée dans la propriété "db.user"

# Vérification

- De façon analogue à GNU Make, on peut vérifier l'existence d'un fichier:

```
<target name="check_jar">
  <available file="${file}" property="found"/>
  <antcall target="check_message"/>
</target>

<target name="check_message" unless="found">
  <echo message="Could not find ${file}"/>
</target>
```

antcall sert à appeler explicitement une autre cible

# Ant vs. Maven vs. Gradle

- Ant: très flexible, utilise XML pour les scripts
  - Pas de règles sur la structure des projets
- Maven: utilise toujours XML mais impose une structure de projet
  - Plugins dédiés pour des tâches spécifiques (ex: dependencies plug-in)
- Gradle: inspiré par Ant (structure) et Maven (plugins), centré sur la gestion de dépendances, langage spécifique
  - Fichiers plus clairs à lire
- Ant:target = Maven:phase = Gradle:tasks

# Maven: exemple HelloWorld

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4     http://maven.apache.org/xsd/maven-4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6   <groupId>baeldung</groupId>
7   <artifactId>mavenExample</artifactId>
8   <version>0.0.1-SNAPSHOT</version>
9   <description>Maven example</description>
10
11   <dependencies>
12     <dependency>
13       <groupId>junit</groupId>
14       <artifactId>junit</artifactId>
15       <version>4.12</version>
16       <scope>test</scope>
17     </dependency>
18   </dependencies>
19 </project>
```



# Gradle: exemple HelloWorld

```
1  apply plugin: 'java'
2
3  repositories {
4      mavenCentral()
5  }
6
7  jar {
8      baseName = 'gradleExample'
9      version = '0.0.1-SNAPSHOT'
10 }
11
12 dependencies {
13     testImplementation 'junit:junit:4.12'
14 }
```

## 2. Systèmes distribués

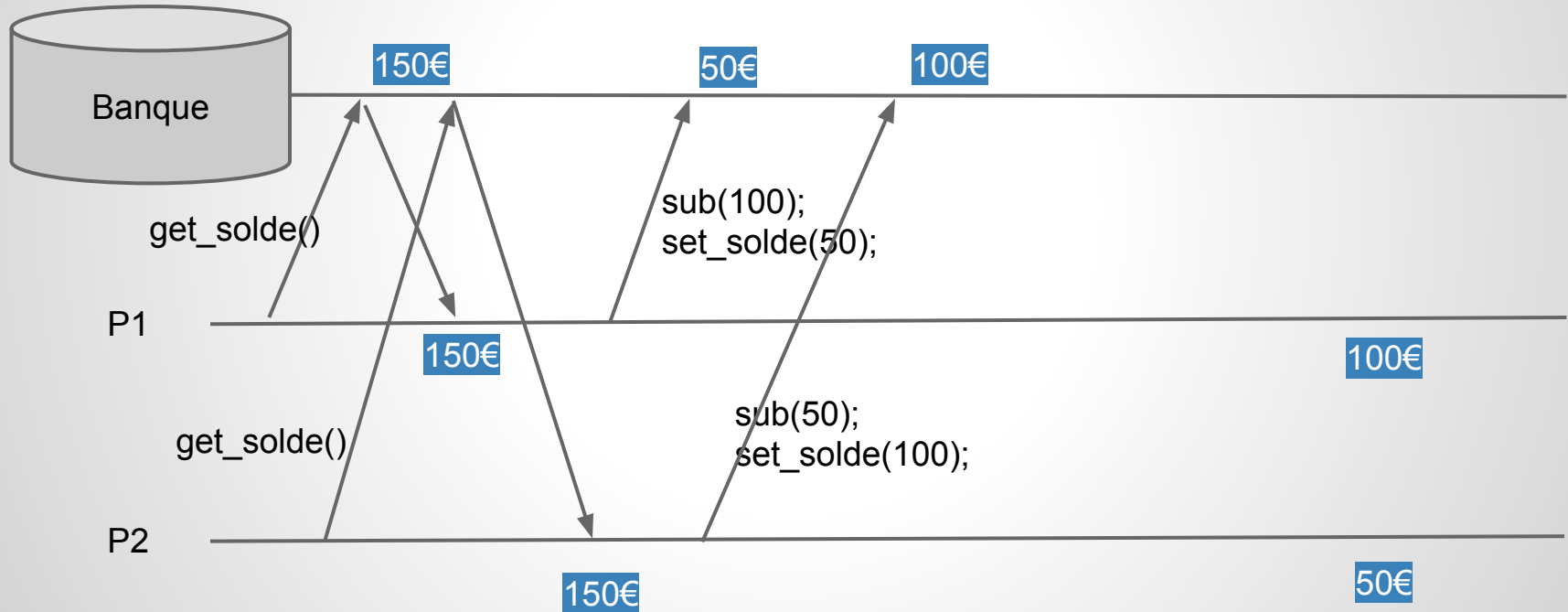
# Systèmes distribués

- Ensembles d'ordinateurs indépendants connectés en réseau et communiquant via ce réseau
- Chaque ensemble apparaît du point de vue de l'utilisateur comme une entité unique
- Mais ils ne partagent pas ni mémoire ni leur horloge
  - Nécessité de synchronisation
- Pourquoi on utilise/développe des systèmes distribués?
  - Améliorer la sécurité (grâce à la redondance)
  - Améliorer les prestations (répartition de la charge)
  - Décomposer la logique de l'application (chaque machine se spécialise sur une tâche particulière)

# Synchronisation

- Situation d'accès à une même donnée partagée
- Accès **synchrone**: horloge globale, synchronisation de tous les processus à chaque phase
  - Avoir une horloge globale n'est pas possible dans les SD
- Accès **asynchrone**: pas d'horloge globale, chaque traitement local prend un temps quelconque
  - Solutions: envoi de messages, mutex

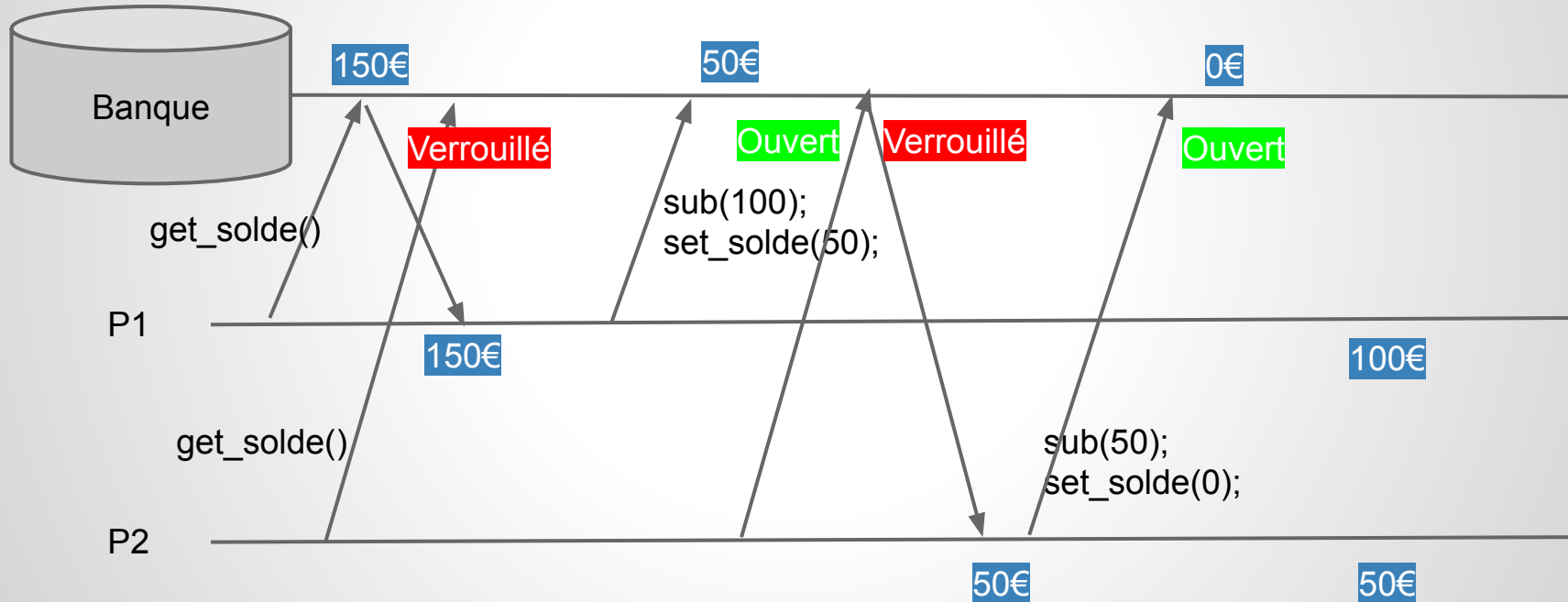
# Modèle asynchrone - exemple



# Mutex (exclusion mutuelle)

- Chaque ressource est munie d'un verrou
- Un processus prend le verrou lorsqu'il désire que personne ne modifie la valeur partagée pendant qu'il effectue un traitement particulier (*section critique*) – Si le verrou est déjà pris: mise en attente du processus
- Il rend le verrou lorsque la section critique est finie

# Modèle asynchrone - avec mutex



# Problèmes

- **Inter-blocage (deadlock)** : Un système asynchrone est en inter-blocage si aucun des processus, qui le composent, ne peut progresser
- **Famine (starvation)** : Si, dans un système asynchrone qui n'est pas en inter-blocage, un processus ne peut plus progresser, alors il est en situation de famine

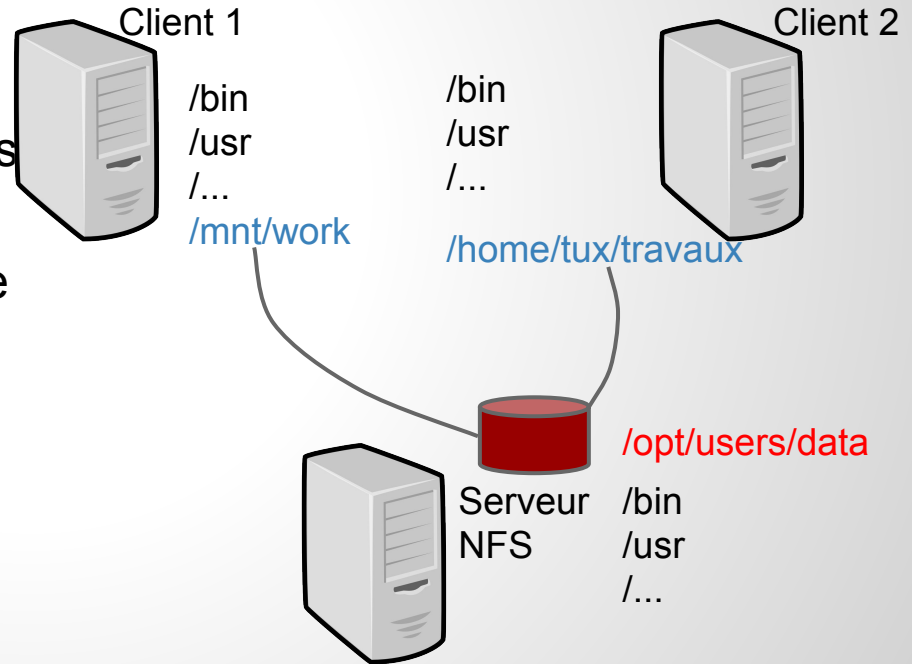


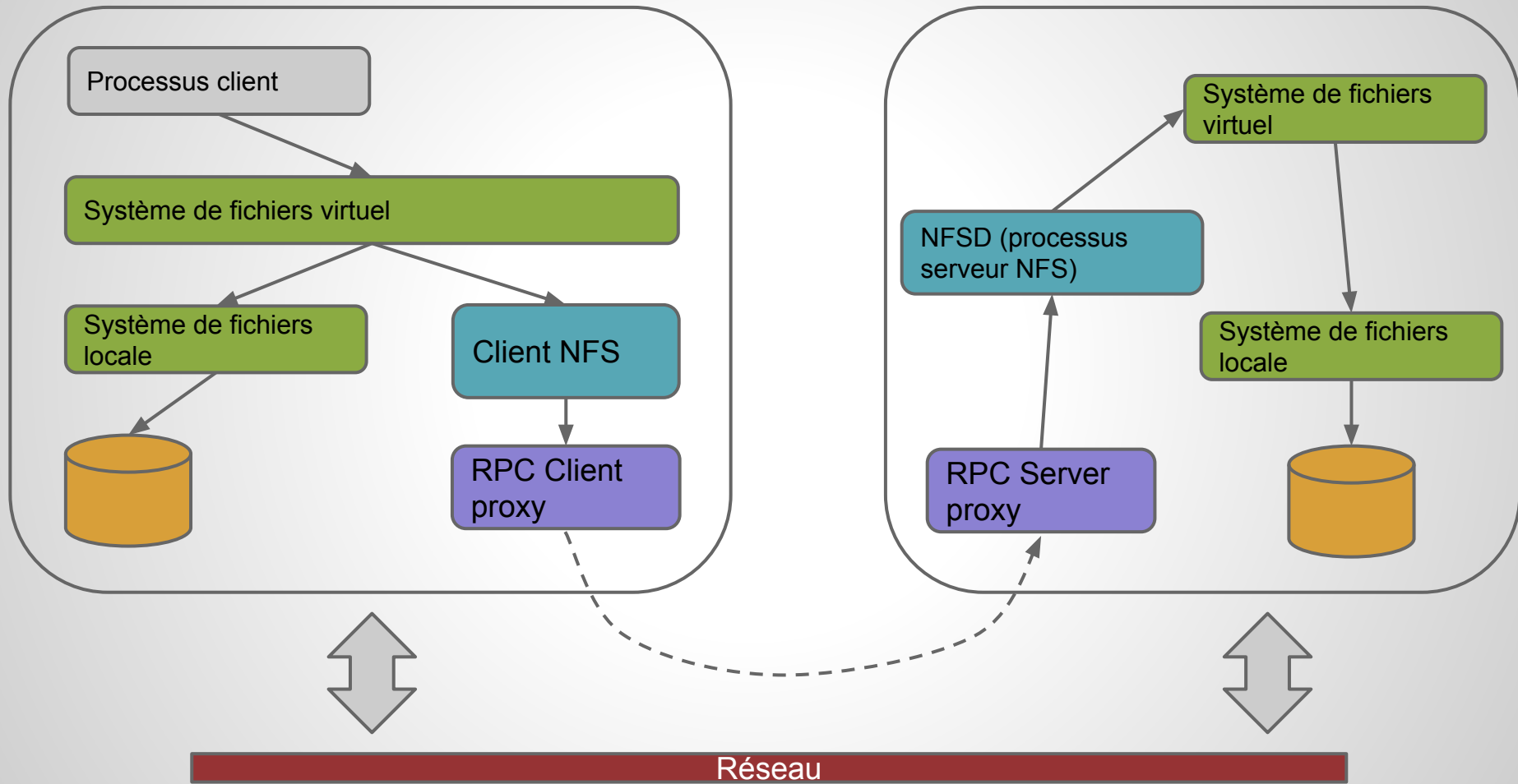
# Propriétés des SD

1. **Transparence à la localisation**  $\Rightarrow$  désignation. L'utilisateur ignore la situation géographique des ressources.
2. **Transparence d'accès**. L'utilisateur accède à une ressource locale ou distante d'une façon identique.
3. **Transparence à l'hétérogénéité**. L'utilisateur n'a pas à se soucier des différences matérielles ou logicielles des ressources qu'il utilise.
4. **Transparence aux pannes** (réseaux, machines, logiciels). Les pannes sont cachées à l'utilisateur.
5. **Transparence à l'extension des ressources**. Extension ou réduction du système sans occasionner de gêne pour l'utilisateur (sauf performance).

# Exemple: NFS - Network File System

- Physiquement : fichiers se trouvent uniquement sur le serveur
- Virtuellement : accès à ces fichiers à partir de n'importe quelle machine cliente en faisant « croire » que ces fichiers sont stockés localement





Les objets répartis:  
Appel de procédure à distance  
(Remote Procedure Calls)

# Appel de procédures à distance (RPC)

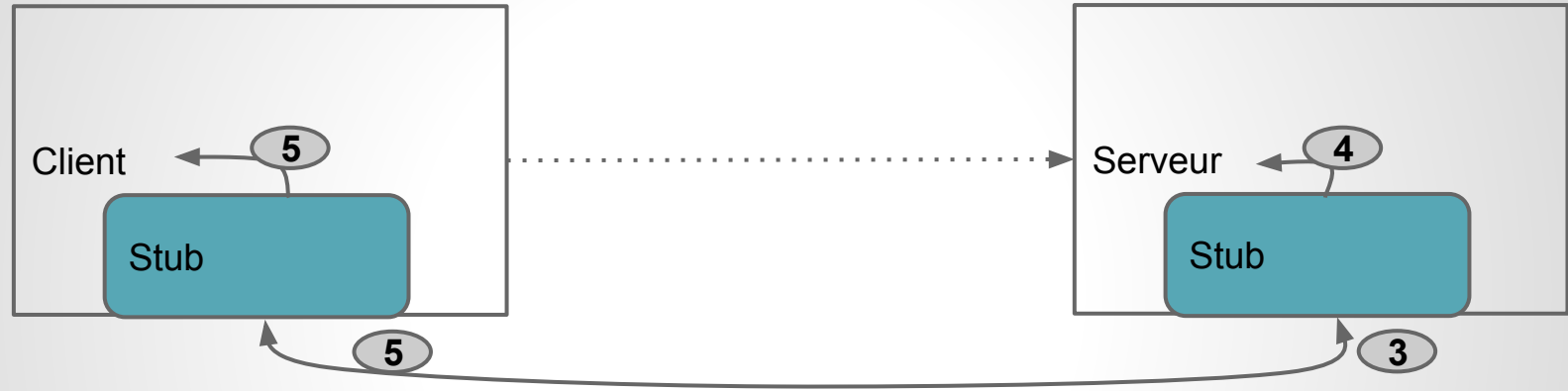
- Mode de réalisation d'une interaction client-serveur
  - ou l'opération à réaliser est présentée sous la forme d'une procédure
  - que le client peut faire exécuter à distance par le serveur.
- Avantages:
  - abstraction de la couche réseaux
  - structure de programme familière pour le programmeur

# Appel de procédures à distance (RPC)



- Apparemment l'appel est fait localement sur l'objet distant
- En effet:
  - 1. Appel d'une procédure locale sur le **stub** (*amorce*)
  - 2. Le stub emballe les paramètres (***marshalling***) et envoie les données nécessaires pour compléter la procédure

# Appel de procédures à distance (RPC)



- Sur le serveur, le stub:
  - 3. déballe (***unmarshalling***) les paramètres
  - 4. renvoie l'appel vers la méthode correcte sur le serveur
  - 5. à la fin de la computation, le résultat est reemballé et renvoyé au client

# Objets répartis

- Extension de l'appel de procédure à distance dans le paradigme OO
- Un objet distribué doit pouvoir être vu comme un objet « normal » (transparence *d'accès*)
- Soit la déclaration suivante :

```
ObjetDistribue obj;
```

- On doit pouvoir invoquer une méthode de cet objet situé sur une autre machine de la même façon qu'un objet local :

```
obj.procedure();
```



# Binding

- On doit pouvoir utiliser un objet distribué sans connaître sa localisation (transparence à la localisation)
- On utilise pour cela un service sorte d'annuaire, qui doit nous renvoyer son adresse

```
obj=ServiceDeNoms.recherche('myDistributedObject');
```

- On doit pouvoir utiliser les objets répartis en tant que paramètres ou valeurs de retour

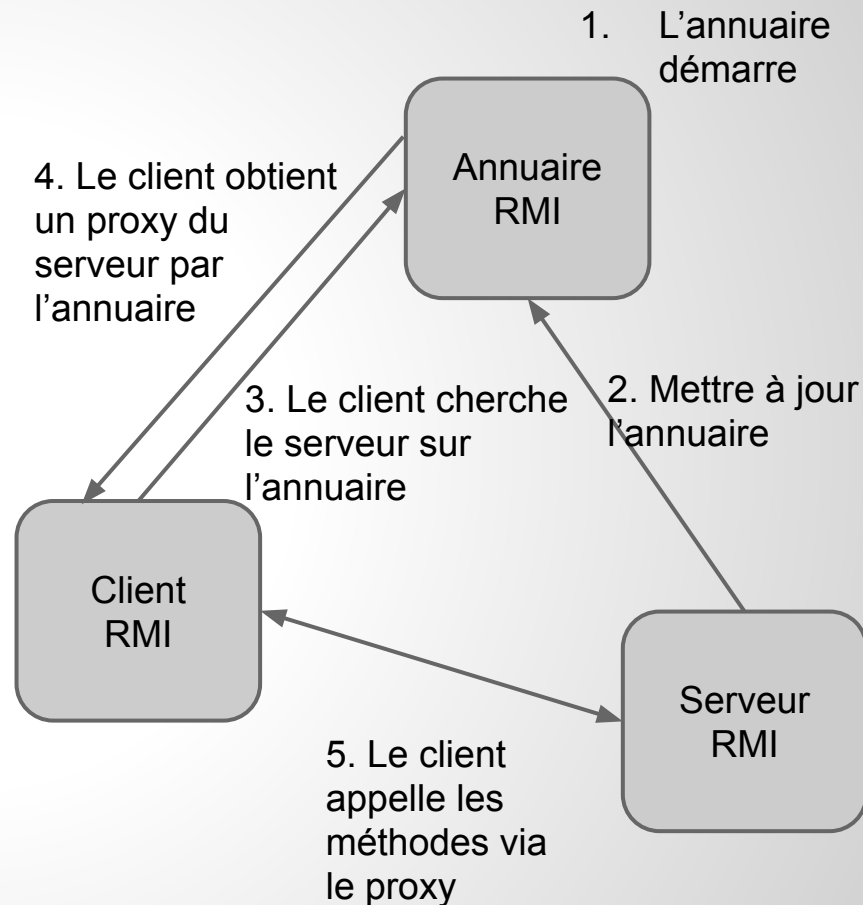
## 3. Java RMI

# Java RMI

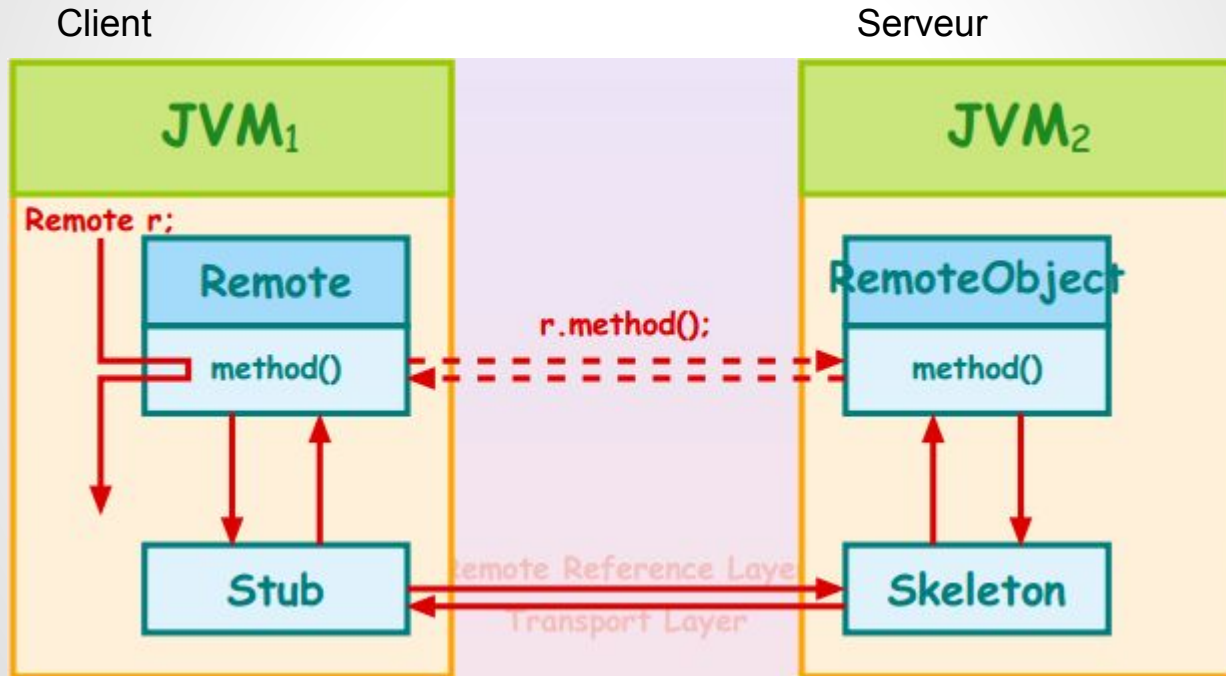
- Implémentation RPC Java
  - (RMI = Remote Method Invocation)
- Permet la communication entre machines virtuelles Java (JVM) qui peuvent se trouver physiquement sur la même machine ou sur deux machines distinctes
- Client/serveur
- Synchrone ou asynchrone (attente du résultat ou non)
- Pas de mémoire partagée

# Java RMI

- Une application RMI est constitué par:
  - Serveur RMI
  - Client RMI
  - *Registry RMI* (annuaire)
- Chaque composante peut être exécuté sur une machine différente



# RPC avec Java RMI



# Définition du serveur

- Quelles méthodes du serveur peuvent être invoquées?
- Quel est le tipage correct des appels distants?
- Il faut définir un service par une interface:
  - Implantée par le serveur
  - Connue du client

# Interface d'un objet distant

- Les seules méthodes accessibles à distance seront celles spécifiées dans l'interface Remote
- Écriture d'une interface spécifique à l'objet, étendant l'interface `java.rmi.Remote`
- Chaque méthode distante doit lever l'exception `java.rmi.RemoteException`
- L'objet distant devra fournir une implémentation de ces méthodes

```
public interface MyRemote extends java.rmi.Remote {  
    public int methode(String s) throws java.rmi.RemoteException;  
}
```

# Implémentation d'un objet distant

- L'implémentation du serveur doit étendre la classe:  
`java.rmi.server.RemoteObject` et implémenter toutes les interfaces qu'il est sensé supporter

```
import java.rmi.*;
import java.rmi.server.*;

public class MyRemoteImpl extends RemoteObject implements MyRemote {
    public MyRemoteImpl() throws RemoteException {}
    public int methode(String s) {
        return s.hashCode();
    }
}
```

Note: `RemoteObject` est abstraite, on utilise habituellement **`UnicastRemoteObject`**



# Les amorces (stubs)

- Réalisent les appels sur la couche réseau
- Réalisent l'assemblage et le désassemblage des paramètres (marshalling, unmarshalling)
- Une référence d'objets distribué correspond à une référence d'amorce
- Les amorces sont créées par le générateur **rmic**

```
rmic MyRemoteImpl
```

# Binding

- Permet d'obtenir une référence d'objet distribué à partir de la référence locale au stub ;
- Cette fonction est assurée grâce à un service de noms: **rmiregister**
  - (qui possède une table de hachage dont les clés sont des noms et les valeurs sont des objets distants) ;
- **rmiregister** s'exécute sur chaque machine hébergeant des objets distants
- rmiregister accepte des demandes de service sur le port 1099

# Développer une application RMI

1. Définir une interface distante (ex: `Xyy.java`)
2. Créer une classe implémentant cette interface (`XyyImpl.java`)
3. Compiler cette classe (`javac XyyImpl.java`)
4. Créer une application serveur (`XyyServer.java`)
5. Compiler l'application serveur
6. Créer les classes stub et skeleton à l'aide de `rmic`
7. Démarrage du registre avec `rmiregistry`
8. Lancer le serveur pour la création d'objets et leur enregistrement dans `rmiregistry`;
9. Créer une classe cliente qui appelle des méthodes distantes de l'objet distribué (`XyyClient.java`)
10. Compiler cette classe et la lancer.

# **Exemple:**

## **Inversion d' une chaîne de caractères**

ReverseInterface.java: interface qui décrit l'objet distribué

Reverse.java : qui implémente l'objet distribué

ReverseServer.java : le serveur RMI

ReverseClient.java : le client qui utilise l'objet distribué

# Fichiers nécessaires

- Côté client:
  - L'interface `ReverseInterface.java`
  - Le client `ReverseClient.java`
- Côté serveur:
  - L'interface `ReverseInterface.java`
  - L'objet (implémentation de l'interface) `Reverse.java`
  - Le serveur d'objet `ReverseServer.java`

# Interface de la classe distante

- Elle est partagée par le client et le serveur
- Elle décrit les caractéristiques de l'objet
- Elle étend l'interface java.rmi.Remote

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface ReverseInterface extends Remote {  
    String reverseString(String chaine) throws RemoteException;  
}
```

# Implémentation de l'objet distribué

```
public class Reverse extends UnicastRemoteObject implements ReverseInterface {  
    public Reverse() throws RemoteException {  
        super();  
    }  
    public String reverseString (String ChaineOrigine) throws RemoteException {  
        int longueur=ChaineOrigine.length();  
        StringBuffer temp=new StringBuffer(longueur);  
        for (int i=longueur; i>0; i--)  
        {  
            temp.append(ChaineOrigine.substring(i-1, i));  
        }  
        return temp.toString();  
    }  
}
```

# Le serveur

```
public class ReverseServer {  
    public static void main(String[] args) {  
        try {  
            System.setSecurityManager(new RMISecurityManager()); //optionnel  
            System.out.println( "Serveur : Construction de l'implémentation  
");  
            Reverse rev= new Reverse();  
            System.out.println("Objet Reverse lié dans le RMIregistry");  
            Naming.rebind("rmi://192.168.1.14:1099/MyReverse", rev);  
            System.out.println("Attente des invocations des clients ...");  
        } catch (Exception e) {  
            System.out.println("Erreur de liaison de l'objet Reverse");  
            System.out.println(e.toString());  
        }  
    }  
}
```

Objet enregistré dans  
rmiregistry



# Le client

```
public class ReverseClient {  
    public static void main (String [] args){  
        System.setSecurityManager(new RMISecurityManager());  
        try {  
            ReverseInterface rev =  
                (ReverseInterface)Naming.lookup("rmi://192.168.1.14:1099/MyReverse");  
            String result = rev.reverseString (args [0]);  
            System.out.println ("L'inverse de "+args[0]+" est "+ result);  
        } catch (Exception e) {  
            System.out.println ("Erreur d'accès à l'objet distant.");  
            System.out.println (e.toString());  
        }  
    }  
}
```

Obtention de la référence

# Compilation et exécution

1. Compiler les sources (interface, implémentation de l'objet, le serveur et le client )
2. Lancer `rmic` sur la classe d'implémentation : `rmic -v1.2 Reverse`
3. transférer `*Stub.class` et `ReverseInterface.class` vers la machine client
4. Démarrer `rmiregistry` sur la machine serveur: `rmiregistry -J-Djava.security.policy=client1.policy &`
5. Lancer le serveur : `java ReverseServer &`
6. Exécuter le client : `java -Djava.security.policy=client1.policy ReverseClient Alice`

## 4. Les systèmes de gestion de versions

# Programmation “en petit”

- Un seul programmeur qui fait tout
  - Tout le code sur une seule machine
- 
- On peut avoir besoin de:
    - partager le code sur plusieurs ordinateurs
    - garder de différentes versions du code
    - faire des back-ups



# Programmation “en grand”

- Une équipe de développement travaille sur un même projet
- Nécessite de partager documentation, code, tâches, bogues, etc...
- Versions différentes du même projet
  - pour différents systèmes d'exploitation
  - pour différents clients
- Problèmes:
  - Résolution de conflits (deux versions différents du même fichier)
  - Qui a fait quoi?
  - Revenir à une version qui fonctionne (s'il y a des bogues sans sortie...)



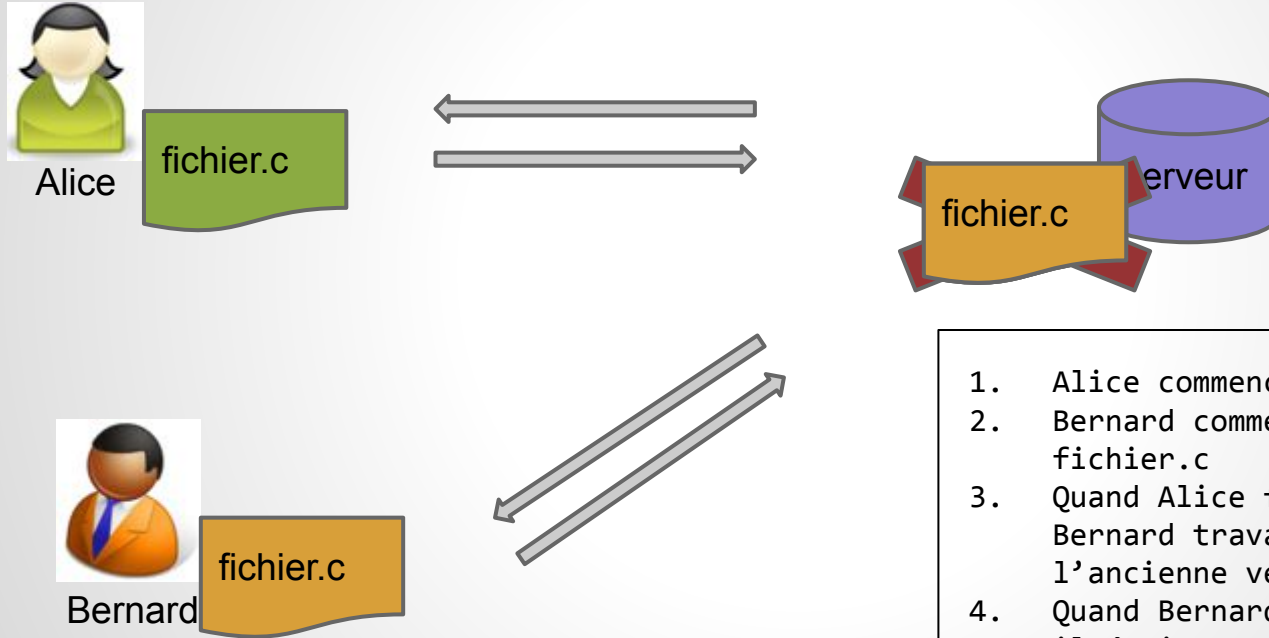
# Systèmes de Gestion de Version

- Systèmes permettant de stocker des informations pour une ou plusieurs ressources informatiques permettant de récupérer toutes les versions intermédiaires des ressources, ainsi que les différences entre les versions
- Fonctionnalités
  - Sauvegarde centralisée des ressources
  - Backup des anciennes versions
  - Résolution des conflits: si plusieurs utilisateurs ont modifié la même ressource
  - Log (journal) des modifications: qui a changé quoi et quand
- Il n'est pas possible de travailler directement sur les fichiers du serveur, il faut travailler sur une copie locale
  - En cas contraire: problèmes d'accès concurrent!

# Strategies de résolution des conflits

- Un conflit est possible si deux utilisateurs vont changer la même ressource en parallèle
- Historiquement, trois solutions ont été proposées:
  - Locking (verrouillage)
  - Merge before commit (fusion avant commit)
  - Commit before merge (commit avant fusion)

# Conflit



1. Alice commence à modifier fichier.c
2. Bernard commence lui aussi à modifier fichier.c
3. Quand Alice fait la sauvegarde, Bernard travaille toujours sur l'ancienne version
4. Quand Bernard sauvegarde son fichier, il écrit sur la version d'Alice
5. Désastre!

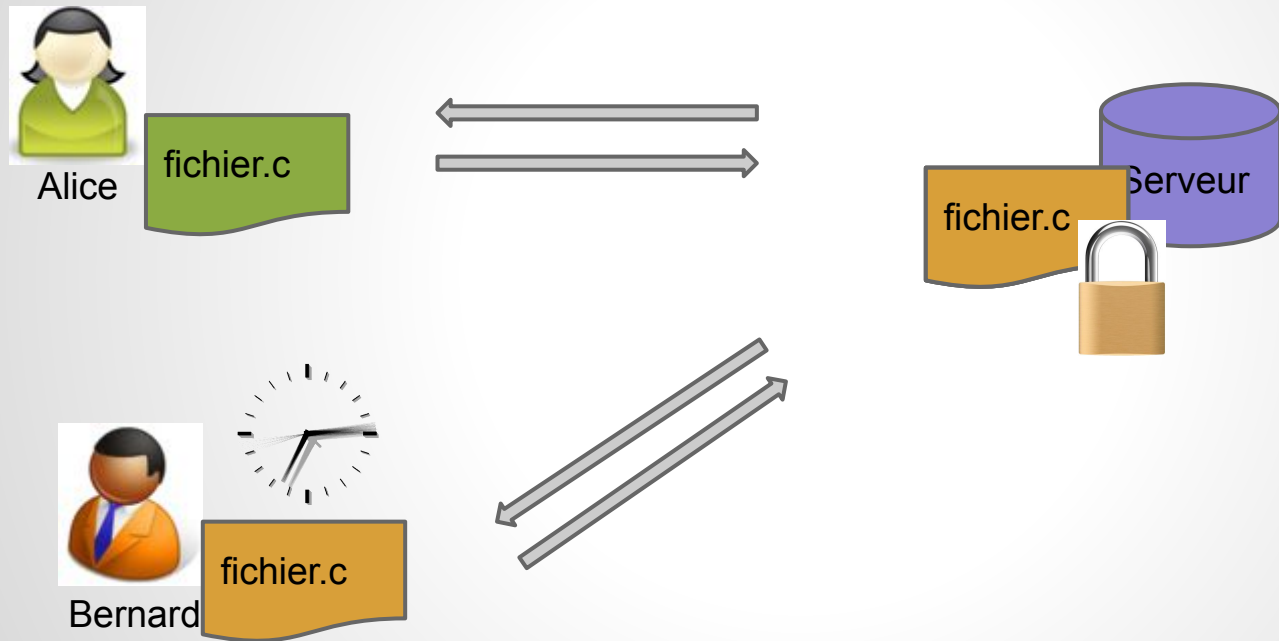


# Verrouillage

- On utilise un verrou sur la ressource qu'on souhaite modifier:
  - Si la ressource a un verrou, personne ne peut la sauvegarder (*commit*)
- Technique qui fonctionne bien si chaque développeur travaille sur des parties différentes du projet, mais c'est rarement le cas

1. Alice verrouille le fichier foo.c et commence à y travailler.
2. Bernard veut travailler sur le fichier foo.c mais il reçoit une notification: le fichier est verrouillé
3. Bernard ne peut pas continuer. Il pense à prendre une tasse de café en tant que le fichier est verrouillé.
4. Alice a fini ses modifications et fait le commit: sauve foo.c qui est deverrouillé.
5. Bernard finit son café et commence à travailler sur foo.c, qui est verrouillé.

# Verrouillage

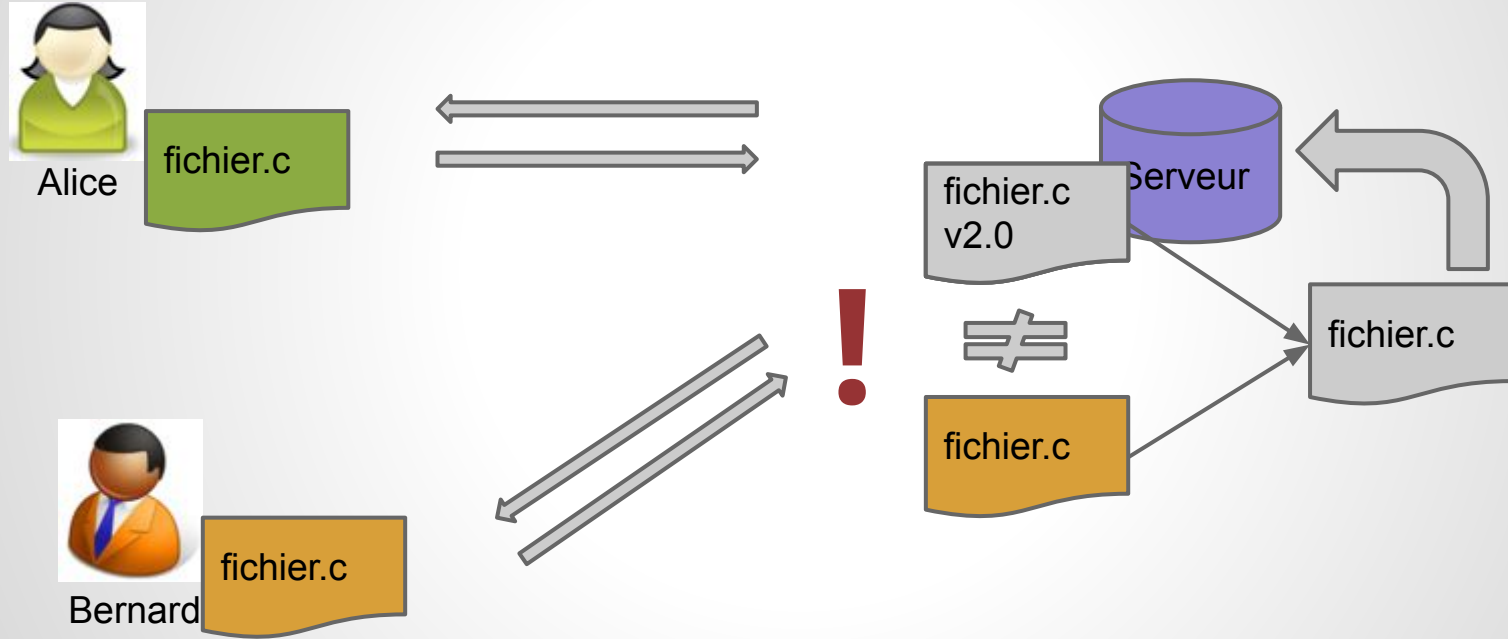


# Merge-before-commit

- Un serveur contient le projet
- L'utilisateur travaille sur une copie locale (l'action de copier un fichier du serveur sur son client est nommé *check-out*)
- L'utilisateur doit résoudre les conflits avant de faire commit
- Technique associée à un modèle client-serveur

1. Alice fait **check-out** du fichier foo.c et commence à le modifier.
2. Bernard fait check-out du fichier foo.c et commence à le modifier.
3. Alice a fini ses modifications et fait **commit**.
4. Bernard essaie de faire le commit mais le serveur l'informe d'un changement de la version dès qu'il avait fait son check-out: il doit résoudre ses conflits (s'il y en a).
5. Bernard fusionne les changements d'Alice avec les siens (**merge**).
6. Le système permet maintenant à Bernard de faire le commit du fichier fusionné

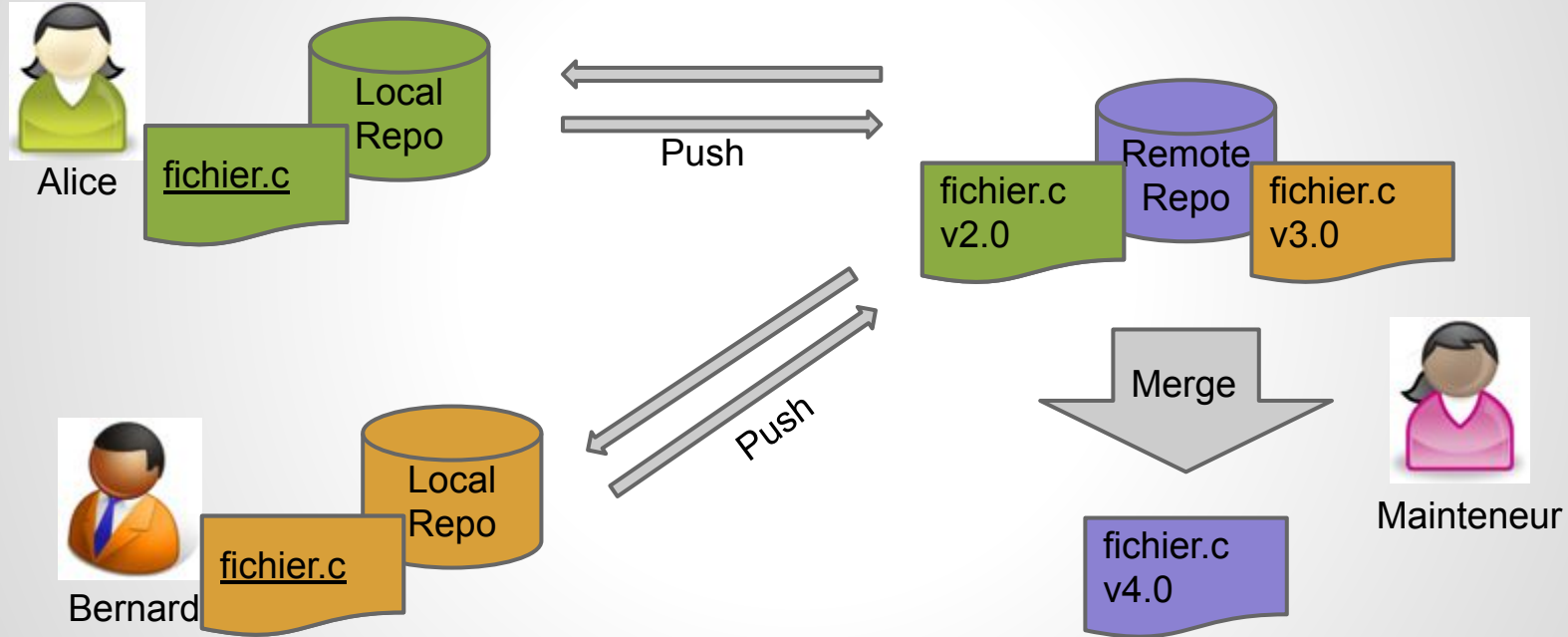
# Merge-before-commit



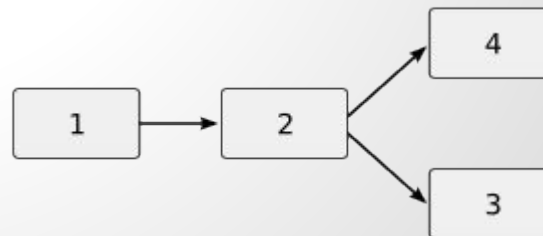
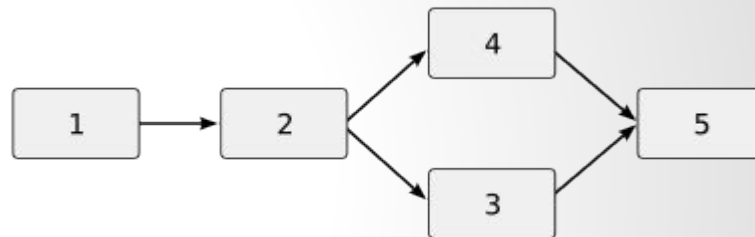
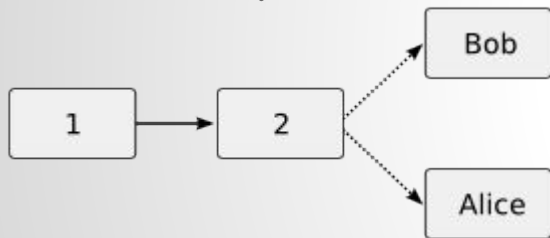
# Commit-before-merge

- Un serveur contient le projet
- L'utilisateur travaille sur une copie locale (l'action de copier un fichier du serveur sur son client est nommé *check-out*)
- Chaque utilisateur a sa version, il fait des commits sur sa copie
- On fusionne les versions dans une étape successive
- Cette technique a le avantage de ne jamais arrêter un commit
- Technique associée à un modèle distribué

# Commit-before-merge



# Commit-before-merge



# Mots-clés

- **Repository** : dépôt de logiciels : un répertoire dans lequel on stocke les informations nécessaires au contrôle des versions d'une série de projets
- **Check-out** : opération d'extraction d'une version d'un projet du repository vers un répertoire de travail local
- **Commit** (ou check-in): action de renvoyer les fichiers au serveur pour les sauvegarder et garder dans l'histoire des modifications
- **Version** (ou revision): un fichier ou un ensemble de fichiers tels qu'ils existaient après une operation de commit
- **Trunk**: la branche principale d'une histoire de révisions
- **Head**: la dernière revision sur la branche principale
- **Working version**: la copie locale du projet
- **Tracking**: action de suivre les modifications d'un fichier



# Subversion: un Système Client-server

# Subversion

- Subversion (SVN) est un logiciel de gestion de versions de type merge-before-commit qui fonctionne selon le modèle client-serveur
- Il y existe aussi une option pour activer le verrouillage
- Le serveur centralisé est unique et contient le repository (dépôt) des fichiers de référence ainsi que le logiciel pour sa maintenance
- Les clients ont une copie du repository et des logiciels clients pour mener les opérations de synchronisation avec le repository
  - Dans Linux: logiciel ligne de commande **svn** (plusieurs interfaces graphiques disponibles)
  - Windows: Tortoise SVN
- Page officielle de Subversion:
  - <http://subversion.apache.org>

# Récupération des données

- La commande **checkout** permet de créer une copie locale d'un repository:

```
svn checkout --username "user" --password "pw" URL_repo
```

- On demande donc ici au serveur de nous envoyer tous les documents sous contrôle de version, dans la dernière version qu'il possède (serveur et *repository* sont renseignés dans l'URL). Comme l'accès est restreint, on précise le login et le mot de passe d'un compte ayant les droits.
- Le résultat de la commande c'est la création d'un dossier qui contient toutes les données du projet

# Rajouter des nouveaux fichiers

- Si on crée un nouveau fichier qui n'est pas inclus dans la version courante, il faut communiquer au serveur qu'on le souhaite rajouter à la liste des fichiers sous contrôle de version (fichiers sous *tracking*). En cas contraire, le commit n'a pas d'effet sur des fichiers hors contrôle
- La commande **add** déclare l'ajout d'un nouveau fichier pour le prochain commit:

```
svn add fichier_a_rajouter
```

- Si l'on ajoute un dossier, “add” va agir récursivement et ajouter toute l'arborescence à partir du dossier.

# Delete/move

- On peut aussi effacer/déplacer des fichiers/répertoires
- La commande **delete** déclare la suppression d'un fichier ou répertoire pour le prochain commit:

```
svn delete fichier_a_supprimer
```

- Pour le déplacement, on utilise la commande **mv**:

```
svn mv fichier_source fichier_destination
```

# Commit

- La commande **commit** enregistre les modifications locales dans le dépôt créant ainsi une nouvelle version.

```
svn commit [-m "message"]
```

- On peut spécifier un message de commit ou pas. Si on ne spécifie pas le message, dans Linux un éditeur de texte est affiché pour y écrire le message. Le message généralement contient des informations pour comprendre la nature de la modification

# Mise à jour

- La commande **update** permet de mettre à jour la working version (copie locale) avec les derniers versions des fichiers présents sur le repository

```
svn update [dossier à mettre à jour] [-r num_revision]
```

- Dans le dossier à mettre à jour (on peut spécifier aussi le nom de dossier en paramètre)
- On peut donner un numéro de revision particulier pour revenir à une certaine version

# Obtenir des informations

- La commande **status** indique les changements qui ont été effectués.

```
svn status [-u -v]
```

- Sans paramètres, la commande montre les ressources qui ont été modifiées depuis le dernier update
- -v: verbose (montre le détail de tous les dossiers et fichiers dans l'arborescence à partir du dossier choisi)
- -u: il compare les éléments avec ceux sur le serveur
  - '': aucune modification ;
  - A : ajouté ;
  - D : supprimé ;
  - C : en conflit ;
  - I : ignoré ;
  - M : modifié ;
  - R : remplacé ;
  - ? : non-versionné ;
  - ! : manquant.



# Gérer les conflits

- La commande **diff** montre les différences entre deux versions.

```
svn diff
```

- Un conflit est déclenché quand on fait un commit d'une ressource dont la version sur le repository a changé depuis notre dernier update
- Deux possible solutions:
  - Contourner le conflit
  - Fusionner

# Gérer les conflits (suite)

- Révenir à la dernière version (les modifications locales sont écrasées):

```
svn revert [fichier]
```

- Fusionner:
  - On fait un update, le serveur montre une liste d'options:
    - Résolution manuelle (e)
    - accepter une fusion automatique (r)
    - Utiliser mes modifications (mc)
    - Utiliser les modifications sur le fichier du serveur (tc)
    - Attendre (p)
- Déclarer la résolution d'un conflit:

```
svn resolved [fichier]
```

# Verrous

- Les commandes **lock** et **unlock** permettent de verrouiller/deverrouiller un fichier

```
svn [lock|unlock] FICHIER
```

Git:  
un SGV distribué

# Git

- Git est un logiciel de gestion de versions de type commit-before-merge qui fonctionne selon le modèle distribué
- Développé par Linus Torvalds (le même de Linux)
- Le repository (dépôt) des fichiers de référence n'est pas unique
- Pas de séparation client/serveur explicite (mais il peut y avoir un repository de référence)
  - Dans Linux: logiciel ligne de commande **git**
  - Windows: msysgit, github
- Page officielle:
  - <http://git-scm.com/>

# Git - fonctionnement

- On trouve une couche supplémentaire par rapport à SVN
  - répertoire de travail ou workspace
  - index ou repository locale
  - répertoire distant
- Le dépôt local est nommé **index**
- L'utilisation standard de Git se passe comme suit :
  - vous modifiez des fichiers dans votre répertoire de travail ;
  - vous indexez les fichiers modifiés, ce qui ajoute des instantanés de ces fichiers dans la zone d'index (répository local);
  - vous validez, ce qui a pour effet de basculer les instantanés des fichiers de l'index dans le repository distant.

# Création d'un repository

- La commande **init** permet de créer un repository:

```
git init [dossier_repo]
```

- Sans arguments, la commande crée un repository dans le dossier courant (on comprend qu'il a été créé si on voit un dossier `.git` dans le dossier courant), sinon le repository est créé dans le dossier en argument
- Pour créer un repository à partir d'un repository distant, on utilise la commande **clone**:

```
git clone [URL]
```

# Tracking des fichiers

- La commande **add** permet de mettre des nouveaux fichiers sous contrôle de version (*tracking*):

```
git add nom_du_fichier
```

- Pour effacer / renommer un fichier, on peut utiliser la commande **git rm** et **git mv**
  - Cela correspond à faire
  - `rm fichier + git commit` pour effacer
  - `mv fichier.old fichier.new + git rm fichier.old + git add fichier.new` pour renommer



# Commit

- La commande **commit** enregistre les modifications du workspace dans le dépôt local (index).

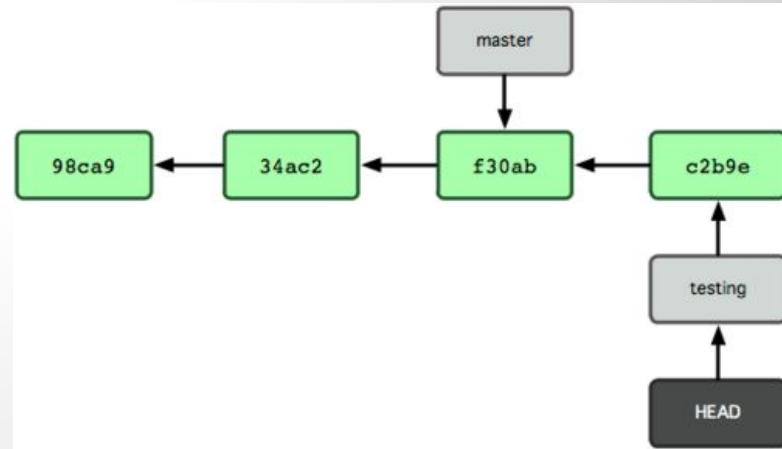
```
git commit [-a]
```

- La commande commit va ouvrir un éditeur de texte pour rajouter un commentaire
- On peut vérifier les modifications à apporter avec la commande **git status**
- La commande **git diff** montre les différences entre votre dernier commit, votre index, et votre répertoire de travail courant
- A difference de Subversion, les fichiers déjà sous tracking ne sont pas rajoutés automatiquement: il faut utiliser l'option **-a** pour cela.

# Gestion des versions dans Git

- Une version dans Git est appelée **branche**
- Il y a deux branches dédiées: **HEAD** et **master**
  - HEAD: un pointeur spéciale qui indique la branche sur laquelle vous vous trouvez
  - master: La branche par défaut. Au

fur et à mesure des validations, la branche master pointe vers le dernier des commits réalisés. À chaque validation, le pointeur de la branche master avance automatiquement



# Naviguer les branches

- La commande **branch** permet de créer une nouvelle branche:

```
git branch nom_de_branche
```

- Pour changer de branche, on utilise la commande **checkout**:

```
git checkout nom_de_branche
```

- Avant de changer de branche, il faut impérativement que tous les changements effectués soient commités ou annulés
- La commande **git branch -a** montre toutes les branches actives

# Naviguer les branches (2)

- Git donne automatiquement un id à chaque commit.
- On peut voir les ids avec la commande **git log**:

```
[davide@darkstar~$] git log  
commit 4d5d7a4224369c957dce3b1e1135bca7b687a27e  
Author: Davide <davide@...>  
Date: Thu Jan 16 22:50:52 2014 +0100
```

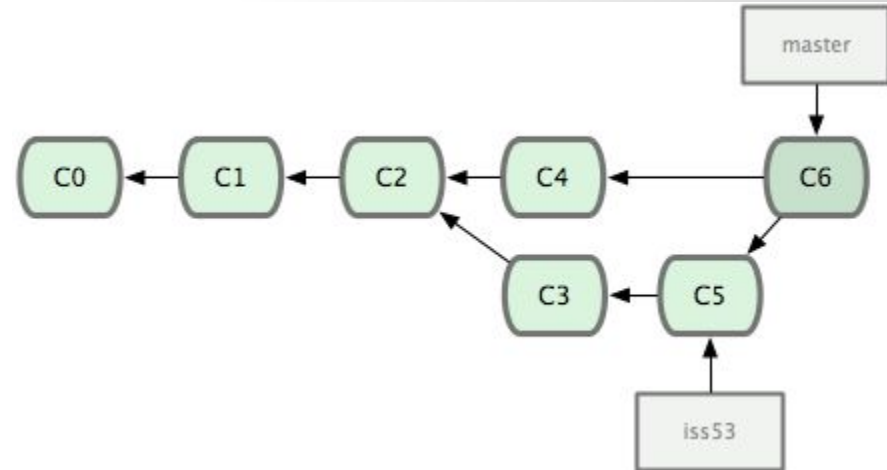
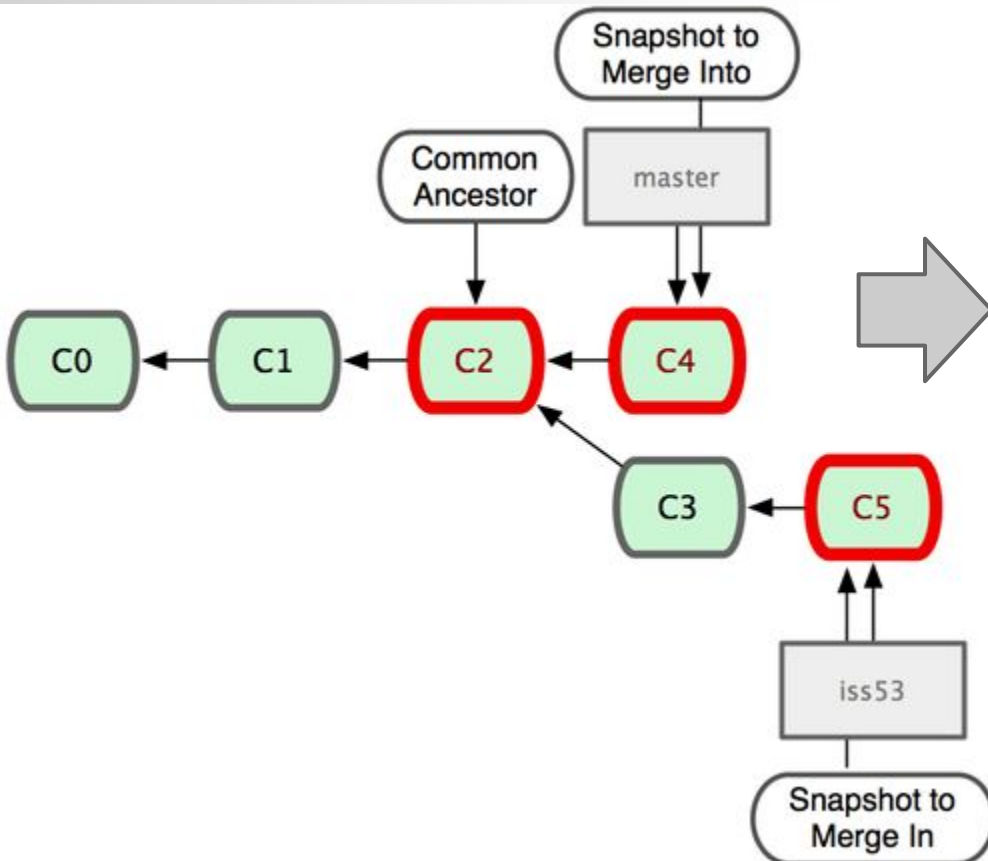
```
test added
```

- On peut positionner HEAD vers un commit spécifique avec la commande **git reset**:

```
git reset [--hard] <id_commit>
```

- Le paramètre <commit> est composé par les premiers 4 symboles du id ou HEAD pour le commit courant
- --hard efface aussi les commits après <id\_commit>

# Fusionner



# Gestion des conflits

- La commande **merge** fusionne la branche courante avec la branche nommée "nom\_branche"

```
git merge nom_branche
```

- S'il y a des conflits, Git laissera un marquage directement dans le fichier, contenant le code de la branche courante, et celui de la branche que vous voulez fusionner. Vous devrez alors corriger le problème manuellement. Une fois corrigé, vous devez commiter les changements pour finaliser le merge.
- Si on souhaite abandonner la fusion, on utilise la commande **reset** :

```
git reset --hard HEAD
```

# Remote

- On peut configurer plusieurs serveur distants avec la commande **remote**

```
git remote add raccourci_serveur URL
```

- La commande **git branch -r** (ou **git remote** dans les anciennes versions) montre toutes les branches remotes
- Un clonage Git (`git clone ...`) vous fournit une branche **master** et une branche **origin/master** pointant sur la branche master de l'origine.

# Mise à jour

- Les commandes **fetch** et **pull** permettent de mettre à jour, respectivement, l'index et le répertoire de travail avec la version du répertoire distant

```
git fetch [branche_distante]  
git pull [branche_distante] [branche_locale]
```

- Fetch télécharge les fichiers du repository distant dans le repository locale, mais les modifications ne sont pas fusionnées jusqu'à un merge
- Pull télécharge les modifications et les fusionne (pull est équivalent à faire fetch+merge)



# Push

- Lorsque vous souhaitez partager une branche avec le reste du monde, vous devez la pousser sur le serveur distant sur lequel vous avez accès en écriture. Vos branches locales ne sont pas automatiquement synchronisées sur les serveurs distants — vous devez “pousser” explicitement les branches que vous souhaitez partager:

```
git push [serveur distant] [branche]
```

