

# Parte 18

## Version control



[G.Klimt – The Kiss, 1907]

# Scambio di modifiche fra sviluppatori

- Quando più programmatori lavorano ad un progetto software, si pone il problema di come scambiarsi le modifiche apportate ad un file

# Soluzione naive

- Scambiarsi l'intero archivio software
  - Se l'archivio è grande, si spreca **banda** di rete per scaricarlo
  - Se l'archivio è grande, si spreca **tempo** per spaccettarlo, configurarlo e compilarlo di nuovo
  - Moltiplicate per il **numero di programmatori** coinvolti
  - L'approccio non scala con la dimensione dell'archivio e nemmeno con il numero di programmatori coinvolti

# Soluzione semi-naive

- Scambiarsi il file in cui sono contenute le modifiche
  - Il consumo di banda di rete diminuisce drasticamente
  - Tuttavia, all'aumentare del numero dei programmatori aumenta la probabilità che un programmatore abbia un file obsoleto
  - L'approccio **non scala** con il numero di programmatori coinvolti

# Soluzione smart

- Scambiarsi le modifiche effettuate ad un file
  - Il consumo di banda di rete diminuisce drasticamente
  - Se le modifiche non sono effettuate sulla stessa porzione di file, possono essere **ricevute ed applicate in parallelo** dai programmatori
  - L'approccio **scala** con la dimensione dell'archivio e con il numero di programmatori

# diff

- Il primo meccanismo usato per scambiarsi le differenze è stato **diff**
  - Programma per UNIX AT&T v5 (1974)
  - usato ancora oggi
- **diff** confronta due file **linea per linea** e restituisce una rappresentazione sintetica delle differenze

# Output di diff

- L'output di diff è noto con due nomi diversi
  - Il nome **diff**, quando si legge l'output per capire le modifiche che si intendono apportare al file
  - il nome di **pezza** o **toppa** (**patch**), quando l'output viene usato per applicare le modifiche
    - Si “mette una pezza” sul file originale e si ottiene la nuova versione

# Formato di una patch

- La **patch** è un file di testo, contenente un insieme di modifiche
- Ciascuna modifica è rappresentata così:
  - Si scrive dove avviene la modifica
  - Si scrive la versione originale della porzione di file
  - Si annotano le modifiche da fare
    - Eliminazione di righe, aggiunta di righe



# Problemi delle prime versioni

- La posizione della modifica era identificata dal numero di riga
  - Schema molto rigido
  - Problemi se un altro programmatore inserisce codice prima della posizione della modifica
- Non si teneva conto di un eventuale cambio di nome del file
- Non si poteva produrre automaticamente la differenza fra insiemi di file in due directory

# Formato “unified”

- Per superare tali limitazioni, è stato introdotto il formato “con contesto unificato” (**unified**)
- Un file di patch contiene più modifiche che trasformano file di partenza in file destinazione
  - Ogni modifica riguarda un file di partenza ed il corrispondente file di destinazione
  - Ogni set di modifiche per coppia di file partenza/destinazione ha una **intestazione**:
    - percorso\_file\_originale data\_modifica*
    - +++ percorso\_file\_modificato data\_modifica*

# Formato “unified”

- L'intestazione è seguita da una serie di blocchi di trasformazione (**change hunk**, o **hunk**)
- Ogni hunk inizia con la specifica sulla parte di codice considerata, espressa con la coppia (l, s) = (linea iniziale, numero di linee seguenti):

**@@ -l,s +l,s @@**

- Il segno '-' si riferisce al file originale, '+' al nuovo
- Segue la specifica di trasformazione (inquadrata nella parte di codice considerata, per individuare più agevolmente la posizione della modifica)

# Specifica di trasformazione

- Il segno '-' all'inizio di una riga indica una riga da rimuovere; '+' indica una riga da aggiungere
- Esempio:

Righe di contesto

...

-riga da eliminare

+riga da aggiungere

...

Righe di contesto

# Esercizio

- Copiare la directory del gestore di sequenze in due directory “**new**” e “**old**”
- Modificare alcuni file della directory “**new**”
- Creare la **patch** che trasforma il progetto “old” nel progetto “new”

`diff -ru old new > old2new.diff`

-r: considera ricorsivamente tutti i file nelle directory

-u: produce la patch nel formato “unified”

I file presenti in una sola delle directory vengono segnalati come inesistenti, a meno di specificare l'opzione

-N → include ogni linea del file nella modifica (con + o -)

# Il comando “patch”

- Dati un file (oppure un albero di file) originale ed una patch, è possibile costruirne la versione modificata
- Si usa il comando **patch** con una delle seguenti sintassi:

```
patch < file_contenente_patch
```

```
patch -i file_contenente_patch
```

```
cat file_contenente_patch | patch
```

# Coerenza della patch

- Se la directory attuale è coerente con le intestazioni nel file di patch → la modifica viene effettuata
- Es.: nella directory `/home/marko`, applico una patch avente un'intestazione `---prj/miofile:`
- Se nella directory esiste un file `/home/marko/prj/miofile` → la patch viene applicata
- Altrimenti, patch esce con un messaggio di errore, producendo un file `miofile.rej` (reject) contenente l'hunk fallito

# Opzione “-p”

- Ogni patch va applicata nella directory corretta (corrispondente a quella di generazione)
- Se invece si vuole applicare una patch generata al di fuori della directory corrente, l'opzione **-p[num]** permette di rimuovere un numero di directory pari a num
- Data un'intestazione `/gnu/src/emacs/etc/NEWS:`
  - p0 → `/gnu/src/emacs/etc/NEWS`
  - p1 → `gnu/src/emacs/etc/NEWS`
  - p4 → `etc/NEWS`
  - p → `NEWS` (di default, rimuove tutte le dir)



# Rimozione di una patch

- È possibile effettuare l'operazione inversa: dati un archivio già modificato ed un file di patch, è possibile ripristinare la versione di partenza
- Si usa l'opzione **-R**

```
patch -R < file_contenente_patch
```

oppure

```
cat file_contenente_patch | patch -R
```

# Esercizio

- Applicare alla directory old la patch `old2new.diff` precedentemente creata
- Rimuovere la patch applicata
- Controllare l'esito delle operazioni
- Quale opzione `-p` viene applicata di default?

# Controllo di revisione

- L'uso manuale di **diff** e **patch** non scala con il numero di modifiche applicate
- È necessario introdurre dei meccanismi di memorizzazione efficiente delle diverse versioni di un progetto software
- **Revision Control System**: software preposto alla gestione delle modifiche su documenti
- Noto anche con il nome di **Version Control**
- Gestisce un deposito di file (repository)
- Consente l'accesso concorrente al deposito da parte di molteplici utenti

# Funzionalità offerte

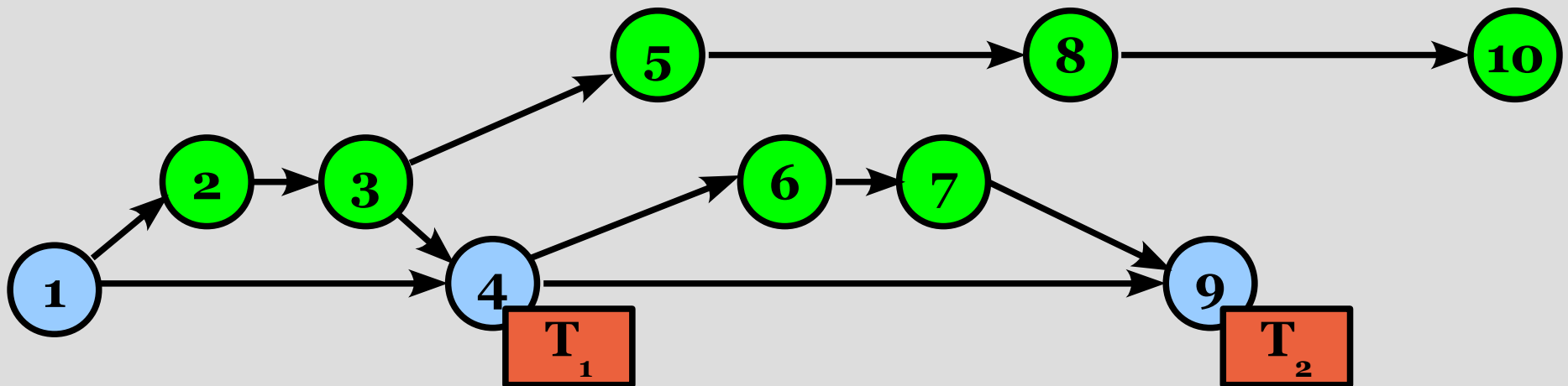
- Inserimento di un progetto software (**import**)
- Scaricamento di una specifica versione del progetto software (**checkout**) in una copia locale (**working copy**)
- Aggiornamento del progetto software alla versione più recente (**update**)
- Inserimento di modifiche locali (**checkin, commit**)
- Cancellazione di modifiche sbagliate (**revert**)
- Marcatura di versioni “interessanti” (**tagging**)

# Funzionalità offerte

- Creazione di versioni “di prova” (**branching**)
- Fusione di una versione di prova con la versione “ufficiale” (**trunk**) del progetto (**merging**)
- Visione della storia del progetto (**log**)
- Produzione di patch fra versioni diverse (**diff**)

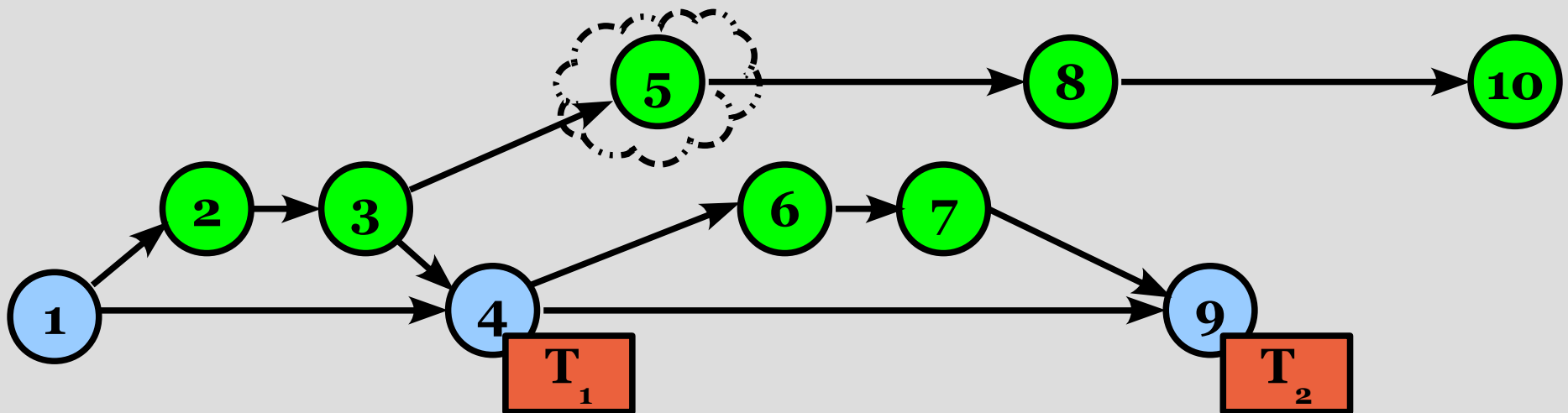
# Rappresentazione

- La storia di un progetto software è rappresentabile con un grafo di modifiche
  - Per convenzione, il grafo si disegna “in orizzontale”



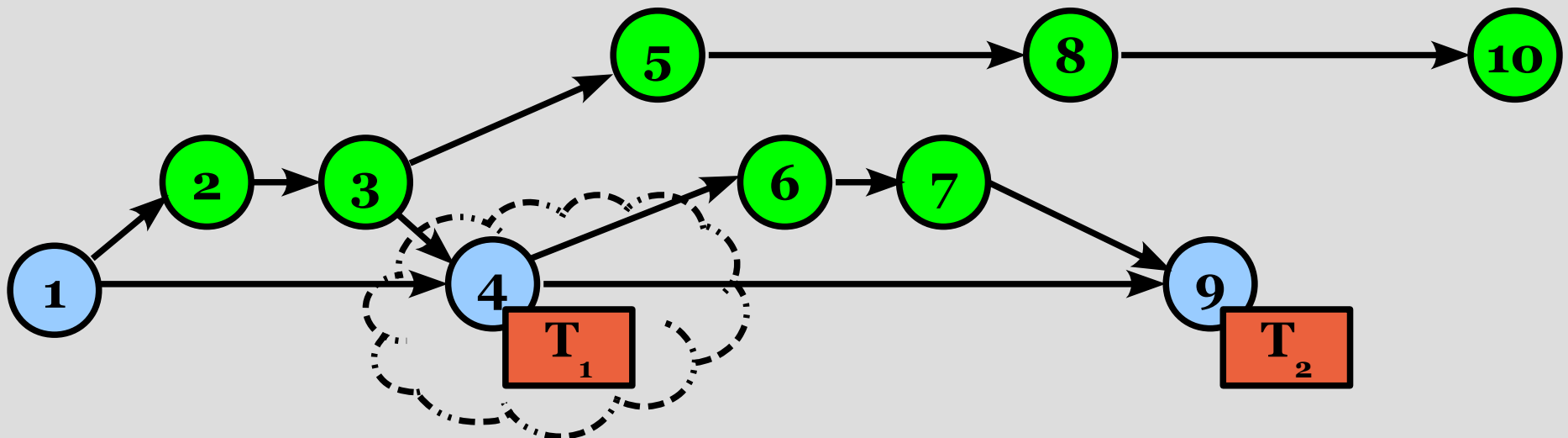
# Commit

- Quello evidenziato è un **commit**, ossia un gruppo di modifiche



# Tagging

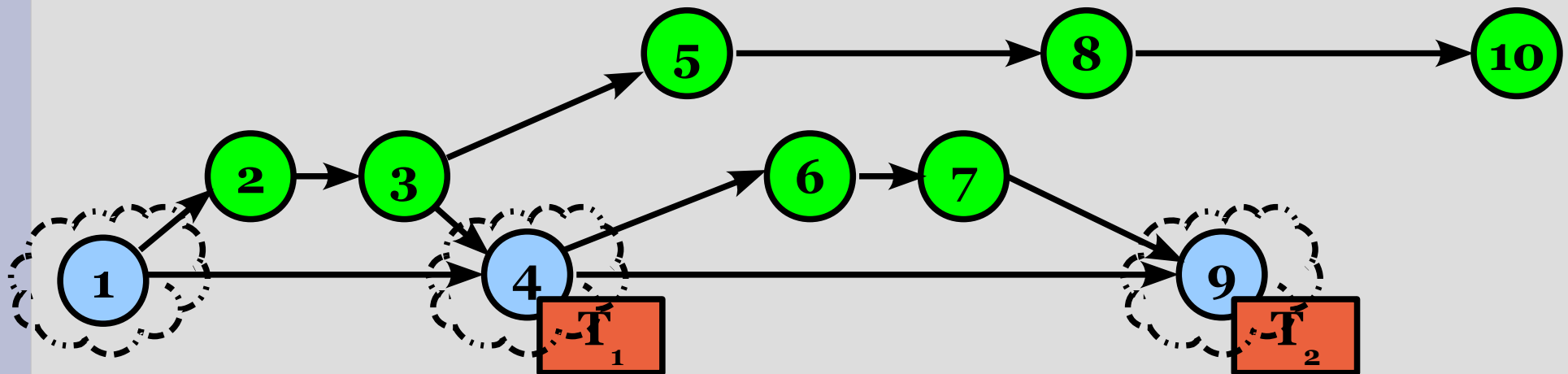
- I nomi dei commit sono spesso indecifrabili
- I commit importanti, rappresentanti ad esempio versioni rilasciate al pubblico, sono corredati di etichette (**tag**) per un accesso più semplice





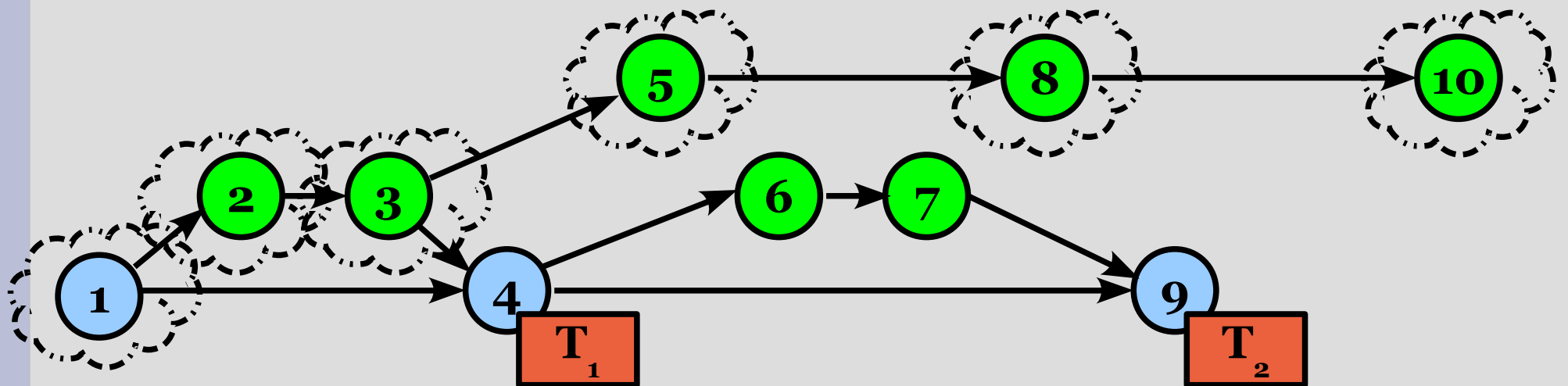
# Trunk

- Una sequenza di commit rappresenta la storia del progetto software
- Qui è mostrato il tronco principale del progetto (**trunk**)



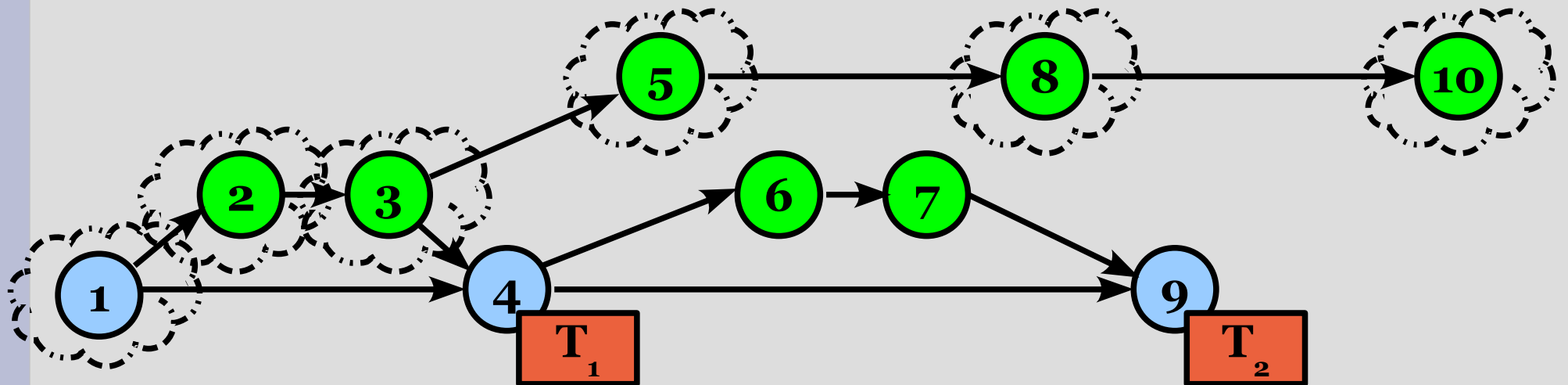
# Branching

- Uno sviluppatore può decidere di tentare alcune modifiche sperimentali, che rivoluzionerebbero il codice attuale
- Si apre una diramazione locale (**branch**) non vista dagli altri programmatori



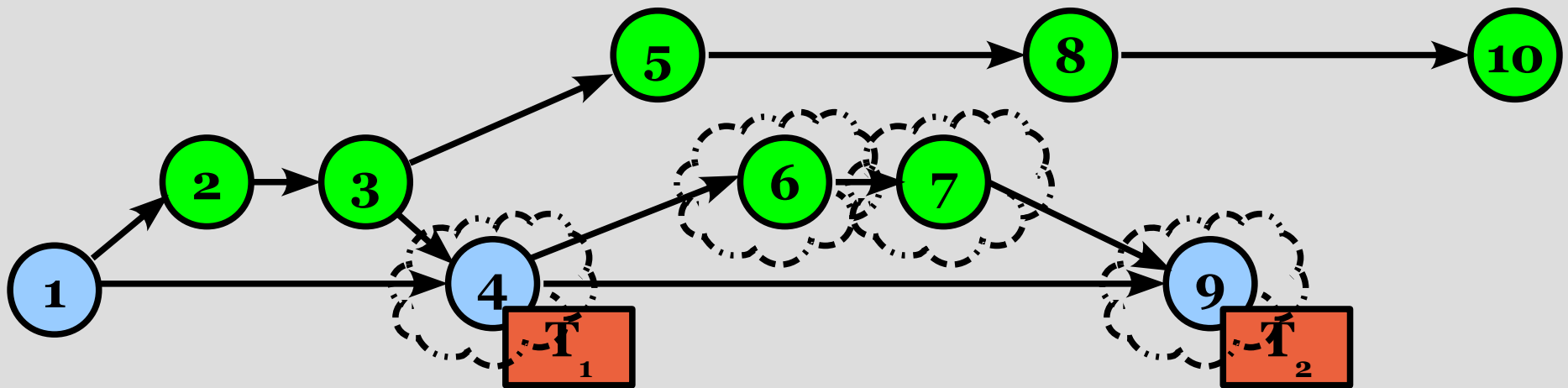
# Branching

- Alcuni branch non hanno futuro e sono lasciati morire così come sono, oppure sono cancellati  
→ Codice **non pronto** per essere incluso nel **trunk**



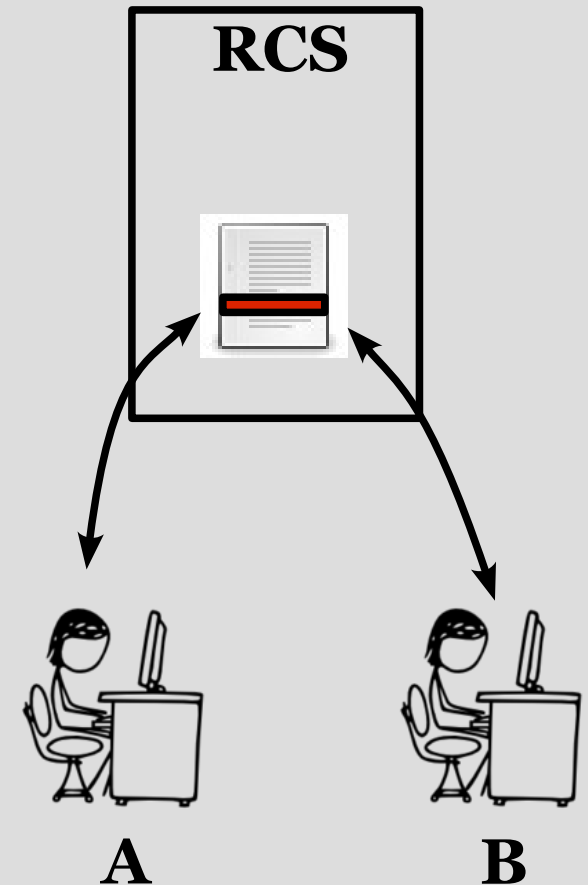
# Merging

- Altri branch sono degni di essere inclusi nel trunk principale
- Si effettua una operazione di fusione (**merge**): le modifiche in 6, 7 sono applicate a 9



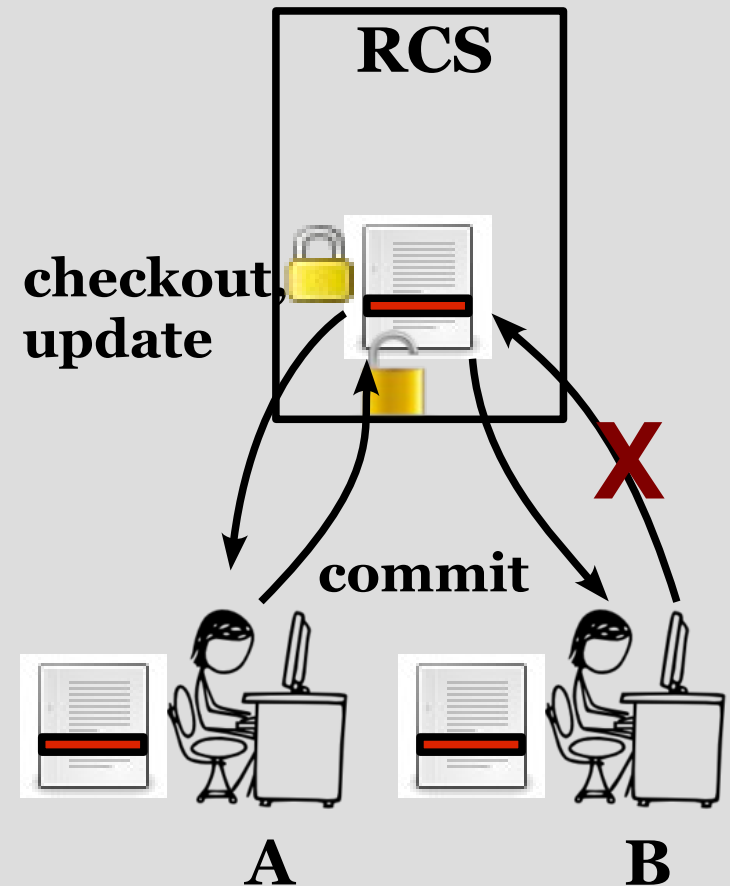
# Gestione dei conflitti

- Una situazione comune nei RCS è il **conflitto**
- Lo sviluppatore A e B vanno a modificare lo stesso file, nello stesso punto
- Il RCS deve segnalare lo stato inconsistente del file ad entrambi gli sviluppatori



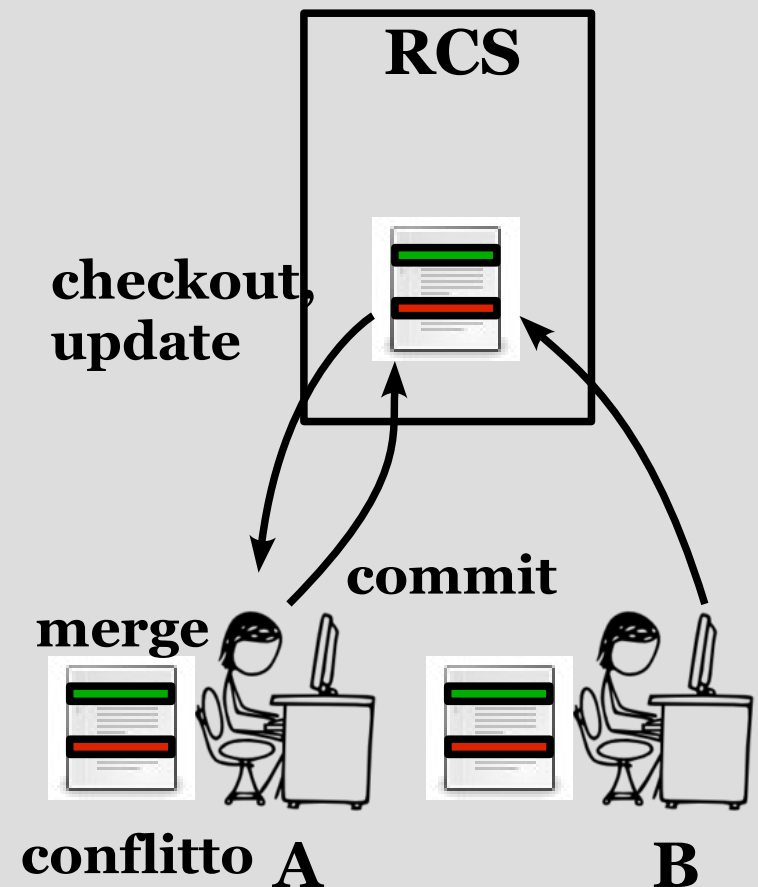
# Lock-Modify-Unlock

- Una soluzione semplice è il **locking del file**
- A prende il lock sul file che intende modificare (**checkout, update**)
- B può leggere, ma non scrivere, il file fino a che il lock non viene rilasciato (in seguito ad un **commit** di A)
- Non scala con il numero di sviluppatori (si creano ritardi nell'accesso al file)



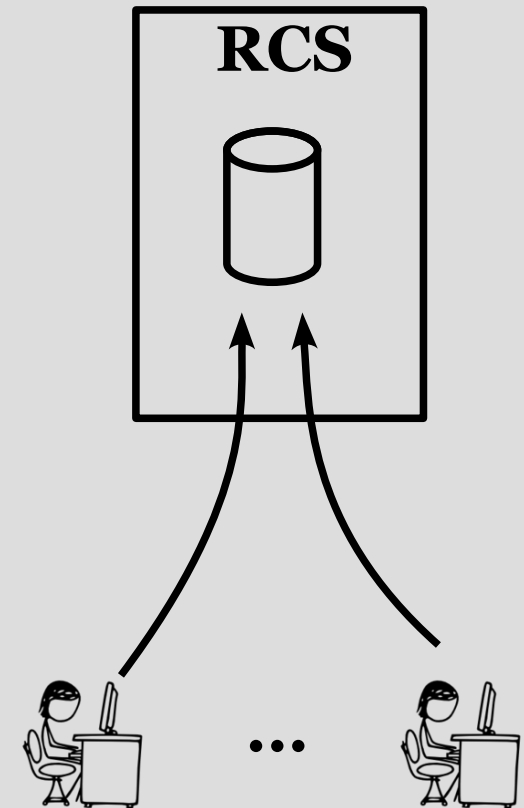
# Copy-Modify-Merge

- Una soluzione migliore è il **merge** delle modifiche ( $\neq$  da merge di un branch)
- A aggiorna la versione del file locale (modificato da B)
- Il RCS aggiorna le modifiche “compatibili” (su righe diverse)
- Il RCS segnala conflitto su modifiche “non compatibili” (fatte sulle stesse righe)
- A risolve il conflitto e ritenta l'aggiornamento del file



# RCS centralizzati

- I primi RCS adottano un modello detto **centralizzato**
  - RCS, SCCS, CVS, SVN
- Un processo server gestisce l'intero albero delle modifiche
- I processi client inoltrano le richieste degli sviluppatori
- Funzionano bene per progetti di medio-piccole dimensioni, con un numero non elevato di sviluppatori





# RCS centralizzati: software

- Revision Control System (RCS)
  - Opera su singoli file (non sull'intero progetto)
  - Sviluppo locale (sullo stesso computer)
  - Lock-based (lock al checkout, unlock al checkin)
- Concurrent Version System (CVS)
  - Approccio client-server
  - Gestione dei conflitti
- Subversion (SVN)

# Subversion (SVN)

- Software di Revision Control open source
- Mantiene compatibilità con il predecessore CVS, estendendolo con varie funzionalità
  - atomic commit, controllo di revisione di tutte le modifiche, inclusi spostamenti, cambiamenti di nome, directory, etc.
- Reference manual:  
<http://svnbook.red-bean.com/nightly/it/index.html>
- Installazione su Ubuntu:  
`sudo apt-get install subversion`

# Moduli di SVN

- **svn**, il client da linea di comando
- **svnadmin**, il programma che permette la creazione e gestione del repository
- **mod\_dav\_svn**, il modulo per Apache che rende disponibile il repository tramite protocollo HTTP
- **svnserve**, il server svn standalone che può – opzionalmente – essere usato al posto del modulo per Apache

# SVN: creazione repository

- Creazione di una SVN repository:  
`svnadmin create /home/marko/newrepos`
- Popola la repository con una serie di directory e file di database:  
`conf/ db/ format hooks/ locks/ README.txt`
- La directory non visualizza il progetto vero e proprio, ma lo codifica in un filesystem virtuale

# SVN: importazione repository

- Supponiamo di voler importare il nostro gestore di sequenze nella repository:

```
svn import gui/progetto_GUI  
    file:///home/marko/newrepos -m  
    "initial import"
```

- Ora il progetto è contenuto nella repository, in una top-level directory del filesystem immaginario
  - Non lo vediamo nella directory della repository vera e propria

# SVN: version control

- Ogni revisione del progetto viene caratterizzata da un **numero** progressivo di **revisione**
- Ad ogni import/commit viene associato un **messaggio di log** che descrive in breve i cambiamenti apportati dalla nuova revisione
  - Tramite l'opzione **-m** “messaggio di log”, oppure tramite un editor che si apre dopo il comando import/commit

# SVN revision keywords

- Esistono chiavi utilizzabili per identificare revisioni “particolari”:
  - HEAD → l'ultima (la più recente) revisione nella repository
  - BASE → la revisione di cui è stato effettuato l'ultimo checkout
  - COMMITTED [file] → l'ultima versione in cui il file specificato è stato modificato
  - PREV [file] → l'ultima versione, prima di quella COMMITTED, in cui il file specificato è stato modificato

# SVN: creazione working copy

- Per creare una copia di lavoro (**working copy**) locale nella directory corrente:

```
svn checkout
```

```
file:///home/marko/newrepos/gui/progetto  
_GUI
```

- E' ora possibile lavorare sulla copia locale, modificando localmente i file/directory



# SVN: tree changes

- Per aggiungere/cancellare file e directory, non basta effettuare la modifica sulla working copy, ma occorre usare comandi appositi:

```
svn add file_o_dir_da_aggiungere
```

```
svn delete file_o_dir_da_eliminare
```

```
svn copy file_da_copiare file_copia
```

```
svn move file_da_spostare file_nuovo
```

```
svn mkdir nuova_dir
```

- L'operazione verrà realizzata sulla repository al prossimo commit

# SVN status

- Per visualizzare i file/dir modificati **localmente sulla WC** dall'ultimo **checkout** (quindi senza contattare la repository):

```
svn status
```

- Vengono elencati i file modificati, con un carattere che identifica il tipo di modifica:
  - A Added (file/dir da aggiungere)
  - D Deleted (file/dir da rimuovere)
  - M Modified (file modificato)
  - ? file/dir non soggetto a revision control

# SVN status -u

- Per visualizzare i file/dir modificati sulla **repository** dall'ultimo **checkout** (in questo caso la repository viene quindi contattata):  

```
svn status -u
```
- I file modificati nella repository dall'ultimo checkout vengono marcati con un asterisco \*
  - Le altre lettere (A,D,M,etc.) si riferiscono invece sempre alle differenze tra la copia locale e l'ultimo checkout

# SVN diff

- L'elenco dettagliato delle modifiche locali viene generato con:  
`svn diff`
- Vengono visualizzate tutte le modifiche realizzate sulla copia locale dall'ultimo checkout realizzato (revisione BASE), in formato compatibile con il comando **patch**
- Per ripristinare un file alla versione precedente la modifica:  
`svn revert file_da_ripristinare`

# SVN diff -r

- Per visualizzare le modifiche locali a partire da una revisione diversa da quella BASE:

```
svn diff -r #oldrev[:newrev]
```

- Confronta la copia locale (o #newrev, se specificato) e la revisione #oldrev. Es:

- `svn diff -r HEAD` → visualizza le differenze tra la copia locale e l'ultima versione nella repository (HEAD)

- `svn diff -r PREV:COMMITTED [file]` → visualizza l'ultimo cambiamento al file committato sulla repository

# SVN: update e commit

- Per aggiornare la propria copia locale con eventuali successive modifiche alla repository principale:

```
svn update
```

- Per integrare le modifiche locali alla repository principale, occorre fare il commit:

```
svn commit [-m "messaggio di log"]  
(senza -m si apre un editor per inserire  
il messaggio di log associato al commit)
```

# SVN: conflitti

- Quando si fa il **commit** di una modifica locale su una parte di codice che nel frattempo (dall'ultimo checkout effettuato) è stata modificata nella repository, si ha un **conflitto**
- Occorre prima fare un **update**
  - Aggiorna la working copy all'ultima versione della repository
  - Segnala i file in conflitto

# SVN update

- Durante l'update, vengono elencati i file della working copy modificati, con un carattere che identifica il tipo di modifica:
  - A Added (aggiunto un file/dir)
  - D Deleted (rimosso un file/dir)
  - U Updated (file aggiornato)
  - C Conflicted (file in conflitto)
  - G Merged (effettuato merge delle modifiche al file)



# Risoluzione dei conflitti

- In caso di conflitto, viene visualizzata una schermata per la risoluzione dello stesso:

```
Conflict discovered in 'miofile.cc'.
```

```
Select: (p) postpone, (df) diff-full, (e) edit,  
        (mc) mine-conflict, (tc) theirs-conflict,  
        (s) show all options:
```

- Di solito conviene innanzitutto visualizzare il conflitto
  - df → visualizza tutto il diff del file in conflitto
  - dc → visualizza solo le parti in conflitto

# Gestione dei conflitti

- In seguito, si può scegliere come risolvere il conflitto:
  - e, l → apre un editor (e) o un programma esterno (l) per la risoluzione del conflitto
  - r → dichiara che il conflitto è stato risolto
  - mf → “mine-full”: accetta solo le modifiche locali
  - tf → “theirs-full”: scarta tutte le modifiche locali
  - mc → “mine-conflict”: accetta solo le modifiche locali che non confliggono
  - tc → “theirs-conflict”: scarta solo le modifiche locali che confliggono

# Gestione dei conflitti

- L'opzione (p) rimanda la risoluzione del conflitto:
  - Il file in conflitto viene marcato con “C”
  - Se possibile, SVN esegue il merge delle modifiche, evidenziando la regione di conflitto nel file attraverso dei conflict marker
  - Vengono creati tre nuovi file:
    - file\_conflikto.**mine**: contiene la copia locale prima dell'update (non esiste se non è avvenuto il merge)
    - file\_conflikto.**r[#OLD\_REV]**: contiene il file ricevuto prima della modifica locale (vecchia revisione)
    - file\_conflikto.**r[#NEW\_REV]**: contiene il file ricevuto dopo l'update (la revisione recente)

# Conflict markers

- Il file della working copy modificato presenta dei conflict marker del tipo:

parte non conflittiva

```
<<<<<<< .mine
```

parte conflittiva come modificata da me

```
=====
```

Parte conflittiva ricevuta dalla repository

```
>>>>>>> .r[#versione]
```

Parte non conflittiva

- Naturalmente, prima di scartare le modifiche ricevute dalla repository, occorre coordinarsi con gli altri sviluppatori

# SVN resolve

- Per risolvere un conflitto che era stato rimandato, occorre invocare:

```
svn resolve --accept [modo] file_conflittivo
```

specificando il modo di risoluzione del conflitto:

`[mine/theirs-full/conflict]` → visti prima

`base` → torna alla versione di base, prima delle modifiche locali

`working` → utilizza la versione corrente del file presente nel working folder (modificata manualmente rimuovendo i marker)

# Repository remota

- Negli esempi precedenti il server SVN era nel nostro filesystem locale (da cui l'uso di [file:///...](#))
  - Eventuali programmatori esterni non potevano accedere alla repository
- Per avere una repository accessibile dalla rete:
  - Creare un SVN server di rete: svnservice, mod\_dav\_svn, etc.
  - Utilizzare SVN server esistenti, es.: Google Code : <http://code.google.com>

# Google Code

- Sito di project hosting gratuito
- Necessita di un Google account
- Supporta controllo di versione con
  - Subversion
  - GIT
  - Mercurial
- Fornisce indirizzo pubblico della repository, es.:  
<https://mioprogram.googlecode.com/svn/trunk/>

# Utilizzo di Google Code

- Es., per effettuare il **checkout** anonimo in tmp:  
`svn checkout https://mioprogram.googlecode.com/svn/trunk/ tmp`
- Per realizzare modifiche, occorre specificare l'account con l'opzione **--username**, es:  
`svn checkout https://mioprogram.googlecode.com/svn/trunk/ tmp --username nome_google_account@gmail.com`
- Viene richiesta una password da inserire la prima volta che si accede con username (Source → googlecode.com password)
- Per specificare gli sviluppatori autorizzati ad effettuare modifiche: Administer → Sharing



# Esercizio

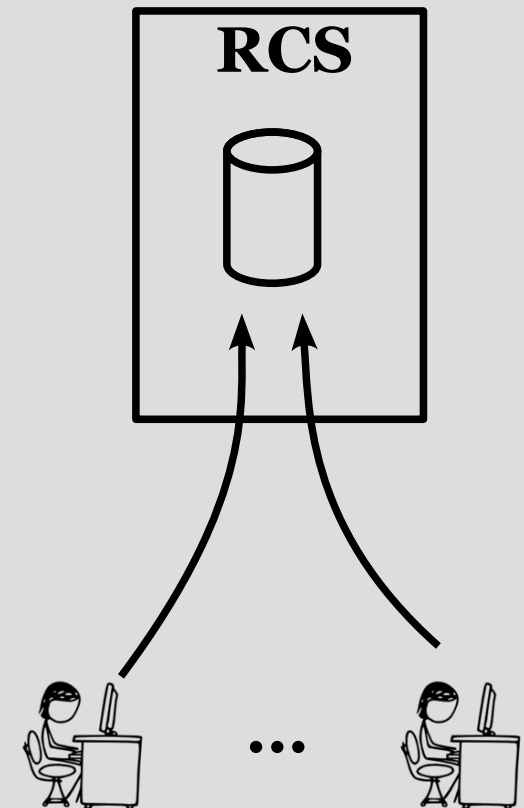
- Provare a creare una repository su Google Code, inserendovi il progetto di gestione delle sequenze
- Eseguire l'**import** e il **checkout** del codice
- Modificare il codice localmente, eseguire il **commit**, e verificare sul sito (Source → Changes) l'esito della modifica
- E' possibile modificare il codice direttamente dal sito (Source → Browse, ..., Edit file), e realizzare il Commit della modifica.

# Esercizio

- Creare manualmente un conflitto in una parte di codice (ad es., modificando il codice sia localmente che dal sito)
- Provare a fare il commit. Funziona?
- Realizzare l'update e provare le varie strategie di risoluzione dei conflitti
  - in particolare, l'opzione (p)=postpone e la modifica manuale della copia realizzata nella working copy

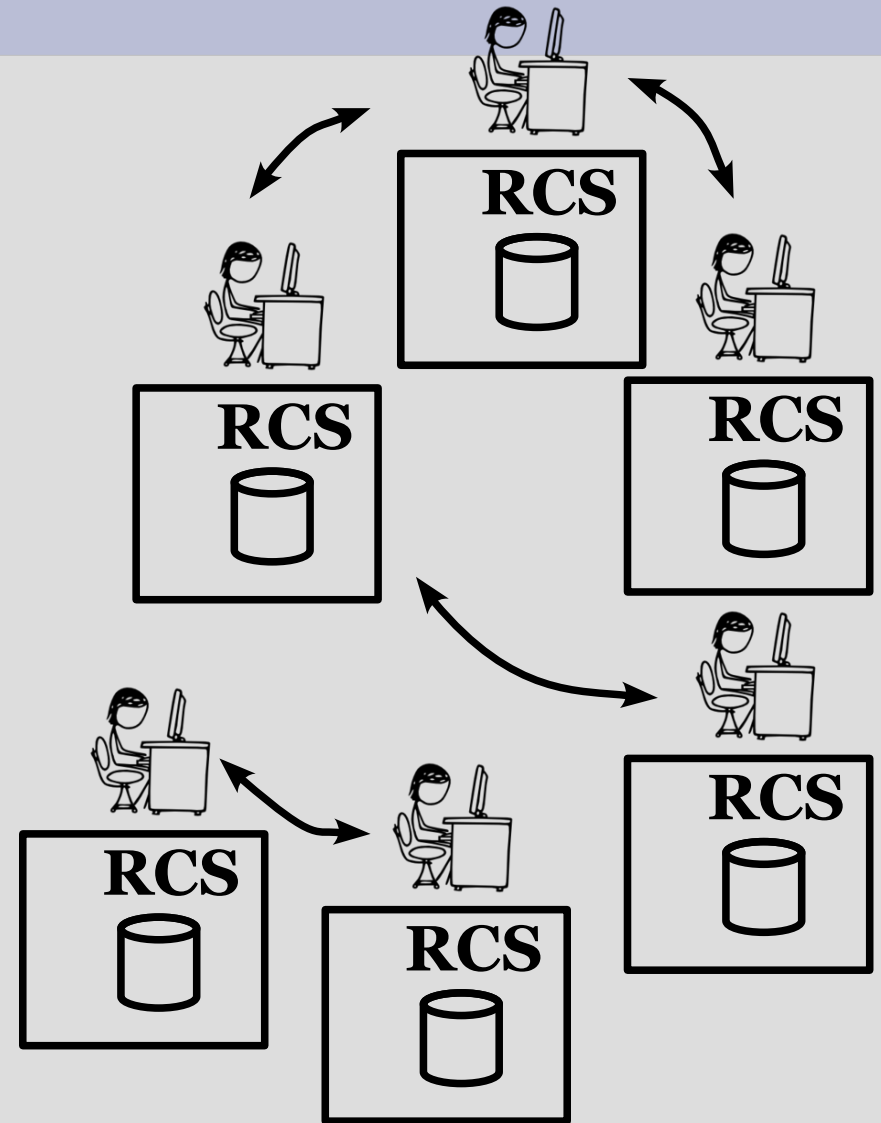
# RCS centralizzati: problemi

- Non scalano con la dimensione del progetto
  - Le operazioni diventano molto più lente
- Non scalano con il numero di utenti
  - Le operazioni diventano molto più lente
  - Un unico sviluppatore non riesce a controllare l'intero flusso delle modifiche



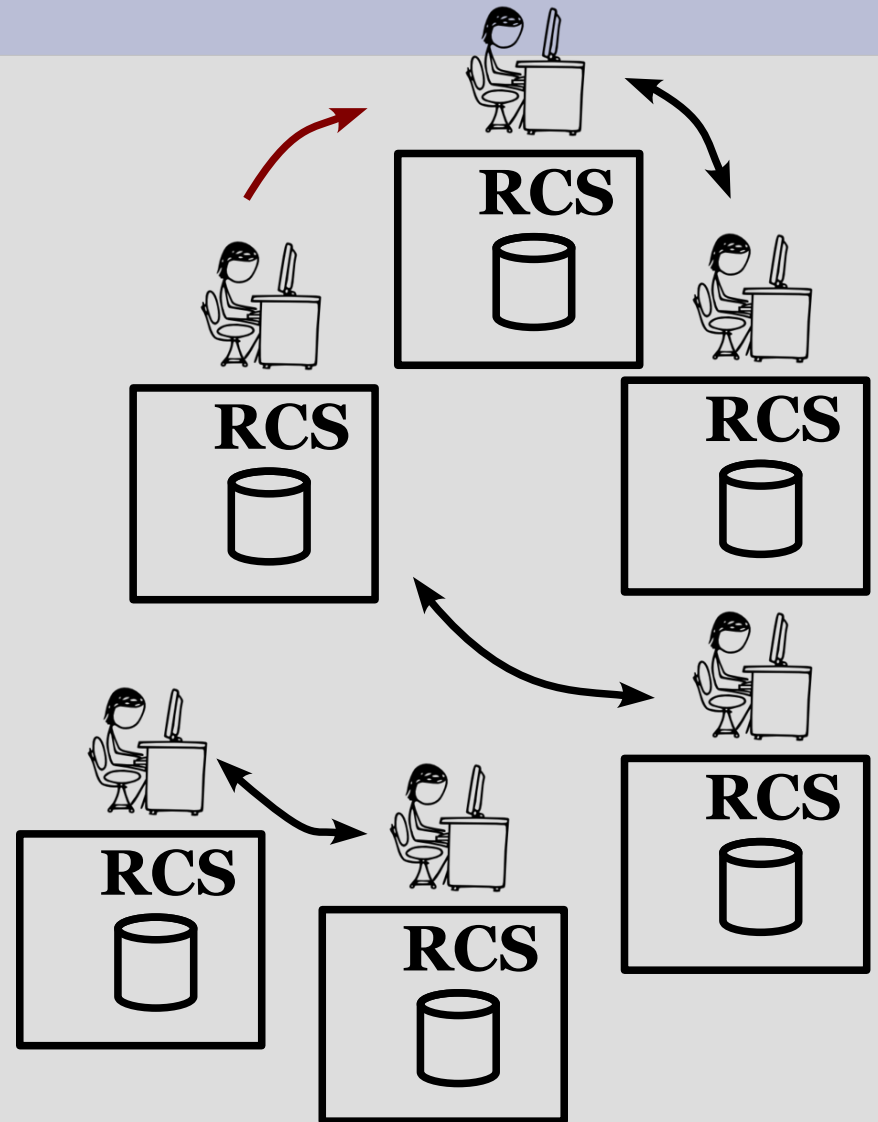
# RCS distribuiti

- Per progetti di medio-grandi dimensioni: modello **distribuito**
  - HG, BITKEEPER, BAZAAR, **GIT**
- Non esiste un singolo repository centralizzato
- Ogni sviluppatore ha una copia della storia del progetto
- La maggior parte delle operazioni avviene localmente



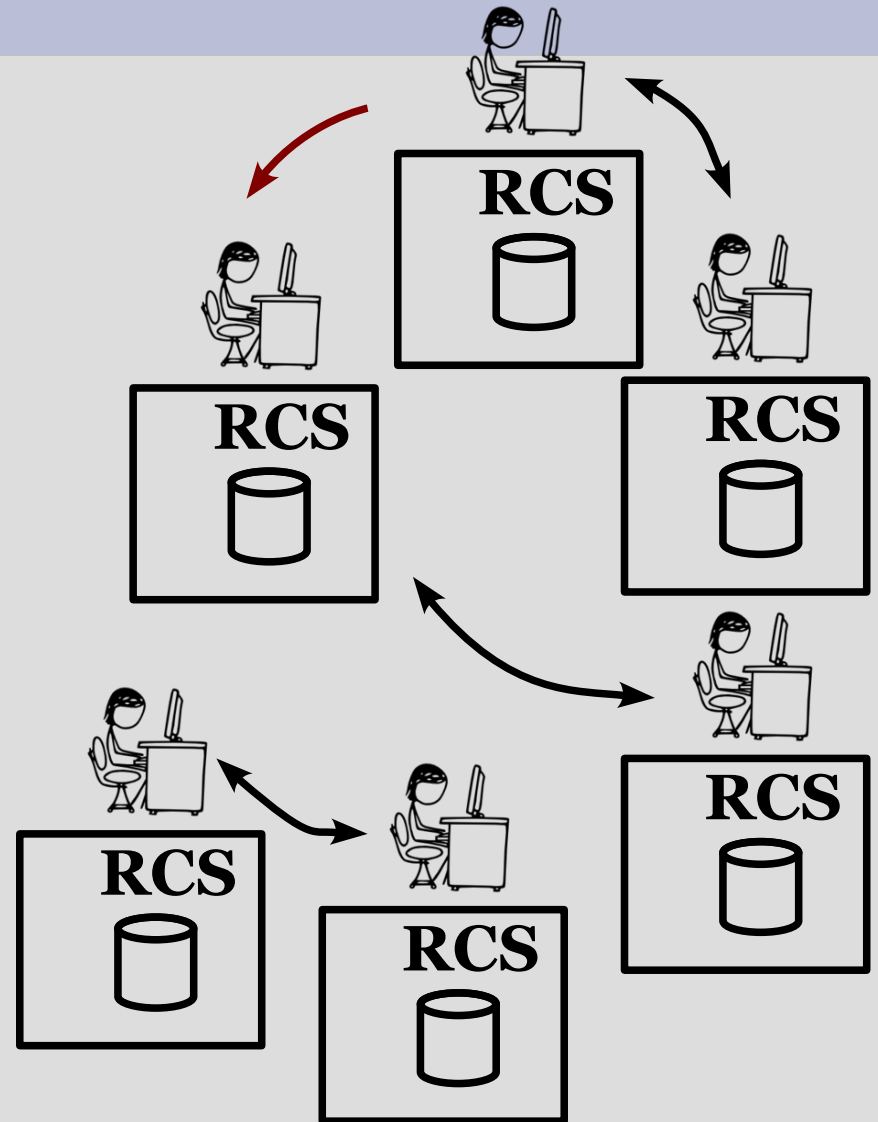
# RCS distribuiti

- Le modifiche effettuate da uno sviluppatore sono spinte (**push**) verso il repository di un altro sviluppatore
  - Comunicazione di rete



# RCS distribuiti

- Le modifiche effettuate da un altro sviluppatore sono prelevate ed importate (**pull**) nella repository locale



# RCS distribuiti

- Invece del modello client-server usato nei sistemi centralizzati, si utilizza un approccio di tipo **peer-to-peer**
- Ogni peer ha una working copy, che rappresenta “in buona fede” il progetto
- Possono esserci più repository “centrali”
- Il merge del codice proveniente da repository diverse avviene sulla base di una “web of trust” → merito storico o qualità delle modifiche apportate

# RCS per il kernel Linux

- 1991-2002: Archivi UNIX, file di patch gestite attraverso la mailing list **linux-kernel**
- 2002-2005: RCS distribuito BitKeeper
  - Ideato da Larry McVoy
  - Proprietario, concesso in uso “free” al kernel
- 2005: Andrew Tridgell, sviluppatore del kernel, prova un reverse engineering del protocollo BitKeeper; per tutta risposta, Larry McVoy ritira la licenza “free”
  - Gli altri RCS esistenti non gestiscono il merge altrettanto velocemente...



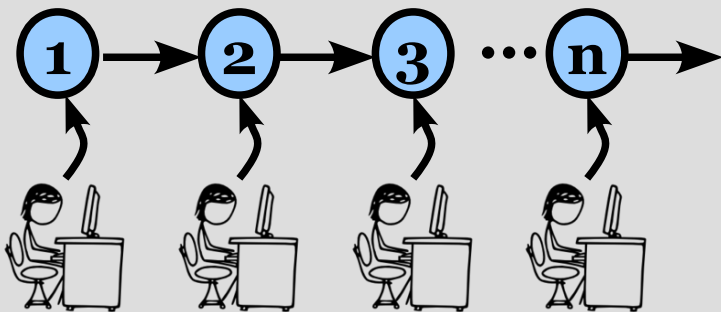
# GIT (the stupid content tracker)

- Aprile 2005
  - Torvalds si ritira per un “sabbatico” di quattro settimane, durante le quali crea **GIT** (**stupido**)
  - RCS distribuito, nato dalle “ceneri” di BitKeeper
  - Vuole essere “stupido” e “veloce”
  - <http://git-scm.com/>
- 2005-
  - GIT è il RCS del codice del kernel (e non solo...)

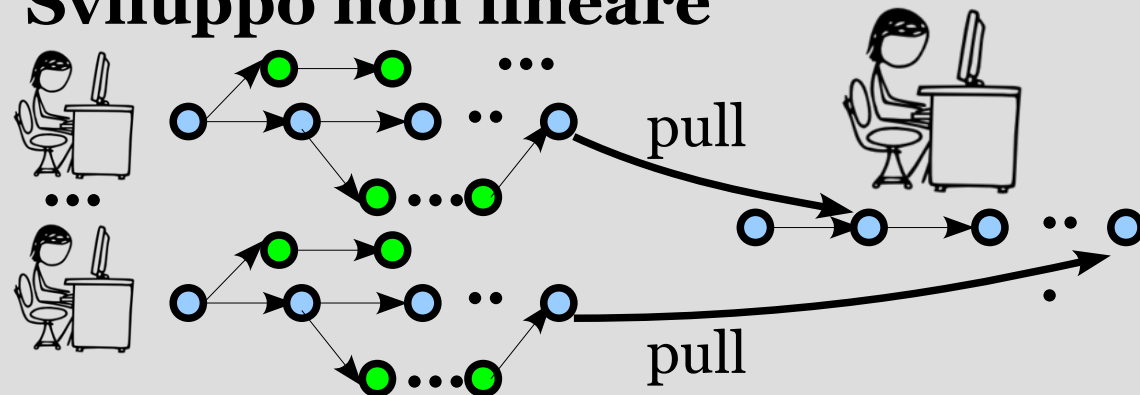
# Caratteristiche di GIT

- Orientato allo “sviluppo non lineare”
  - Lo sviluppo non avviene serialmente in un trunk principale ufficiale, bensì in parallelo nei branch degli sviluppatori
  - Uno sviluppatore “capo” fa il pull delle modifiche ed assembla il suo branch, che può essere visto come trunk principale (convenzione)

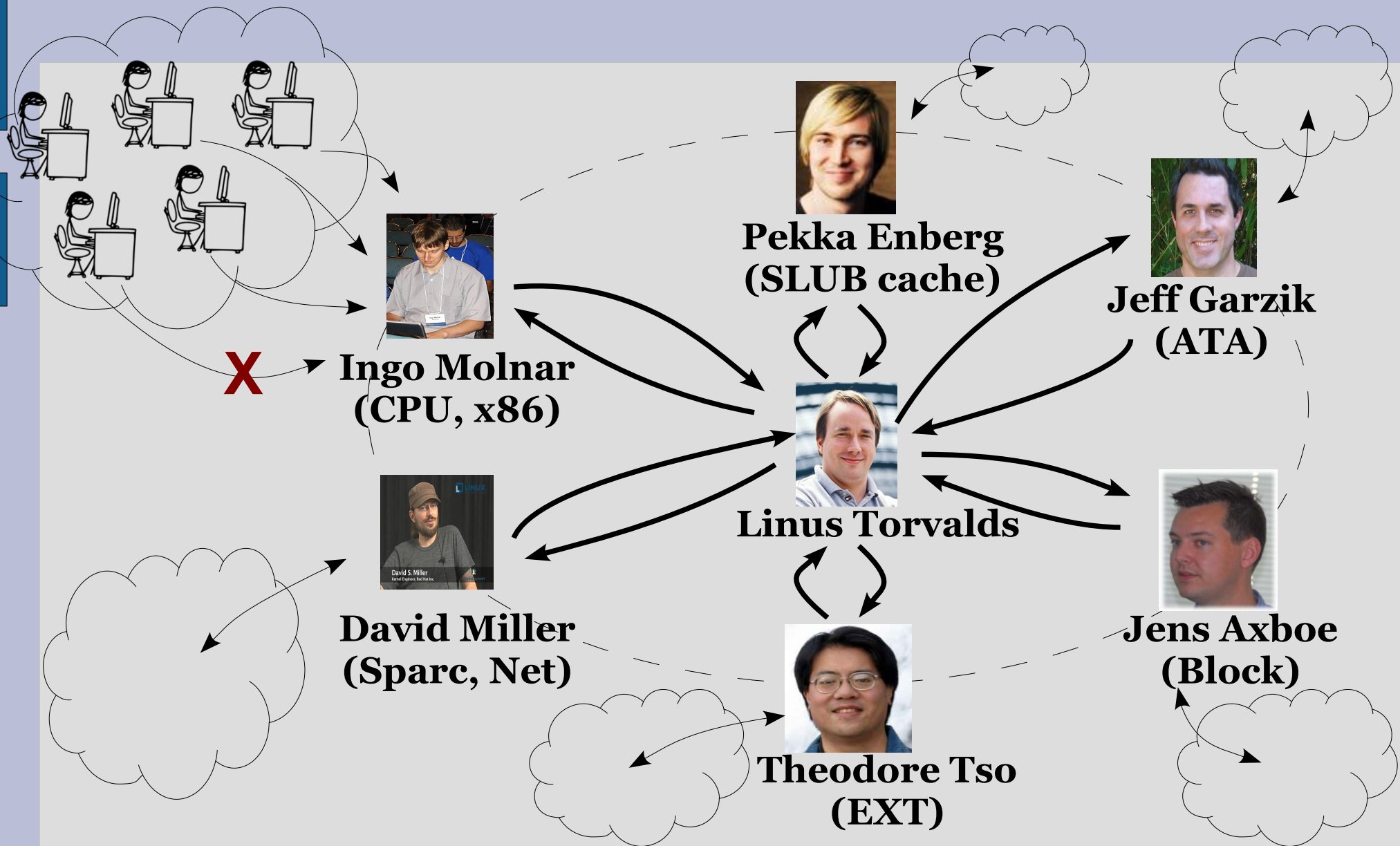
## Sviluppo lineare



## Sviluppo non lineare



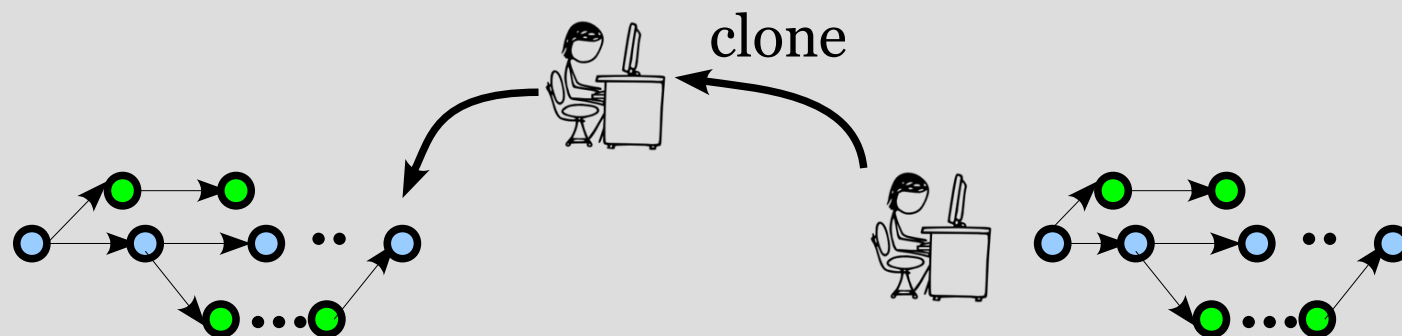
# La relazione fra Torvalds, maintainer ed i collaboratori



# Caratteristiche di GIT

- Orientato allo “sviluppo distribuito”
  - Ciascuno sviluppatore ha un suo branch principale detto **master**
  - GIT rende semplice ed efficiente l'import iniziale di un progetto (**clone**), con la storia completa ed il codice dell'ultima versione in master

## Sviluppo distribuito

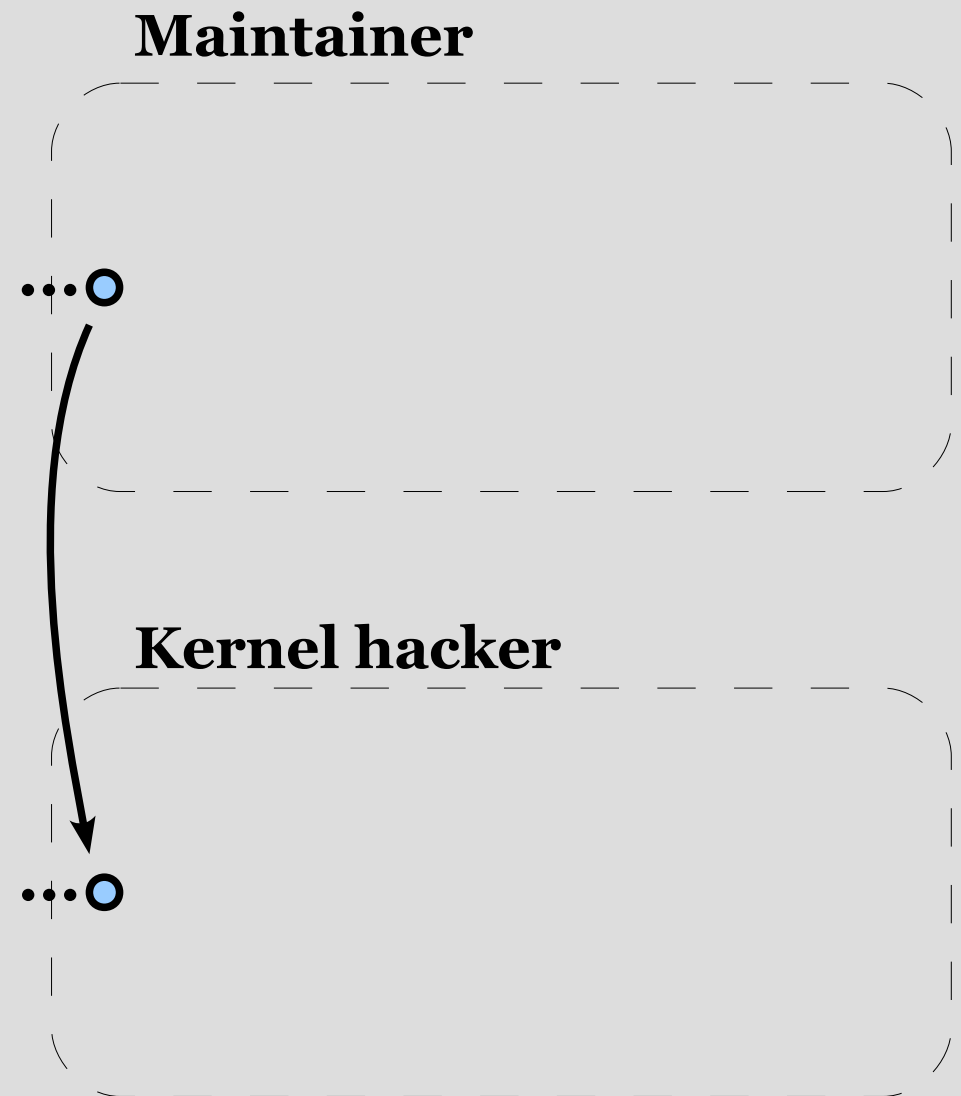


# Caratteristiche di GIT

- Orientato a progetti di “grandi dimensioni”
  - Formato di memorizzazione dei commit molto efficiente e compatto
  - Estremamente veloce (10x rispetto agli altri RCS) nel calcolare le differenze necessarie per portarsi da una versione ad un'altra
- Autenticazione crittografica della storia
  - Il nome originale di un commit è l'hash SHA-1 delle modifiche e della storia precedente del progetto
  - Una modifica ad un commit precedente è immediatamente rilevabile

# Una sessione di un kernel hacker: clone iniziale

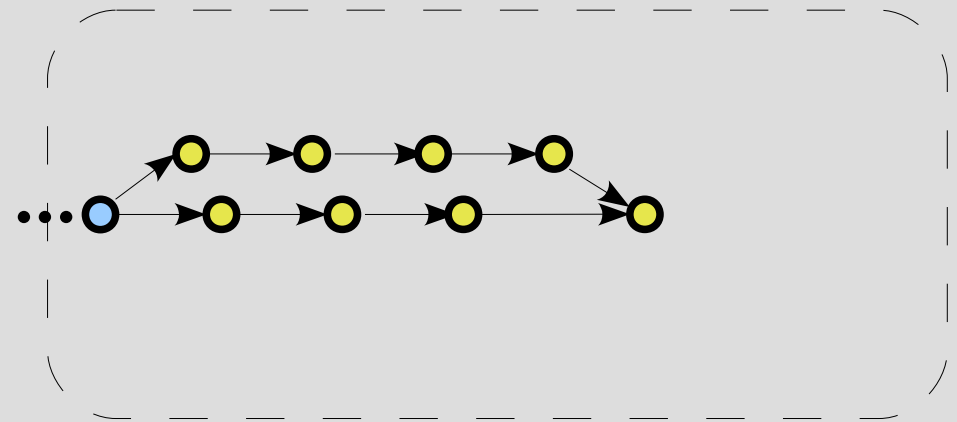
- La prima operazione effettuata è un clone del repository del diretto superiore
  - Il maintainer oppure Linus Torvalds
  - Lo sviluppatore ha una copia aggiornata del branch master



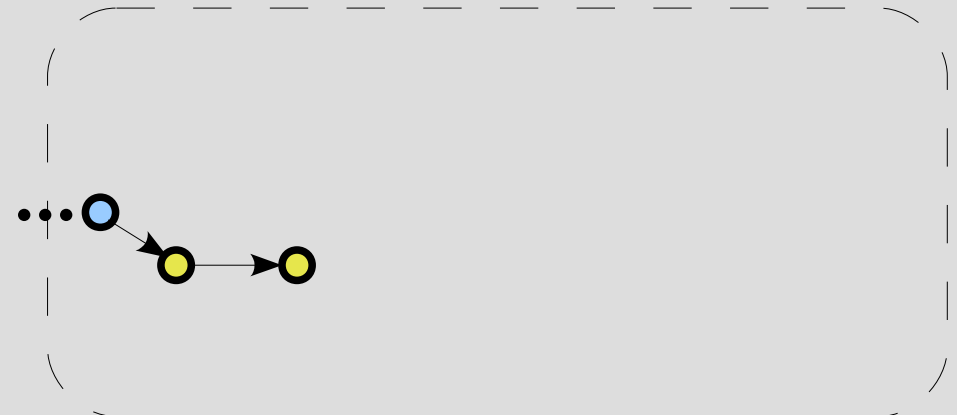
# Una sessione di un kernel hacker: sviluppo non lineare

- I due sviluppatori lavorano in parallelo, producendo nuovi commit, anche in diversi branch
  - Evidenziati in giallo

## Maintainer



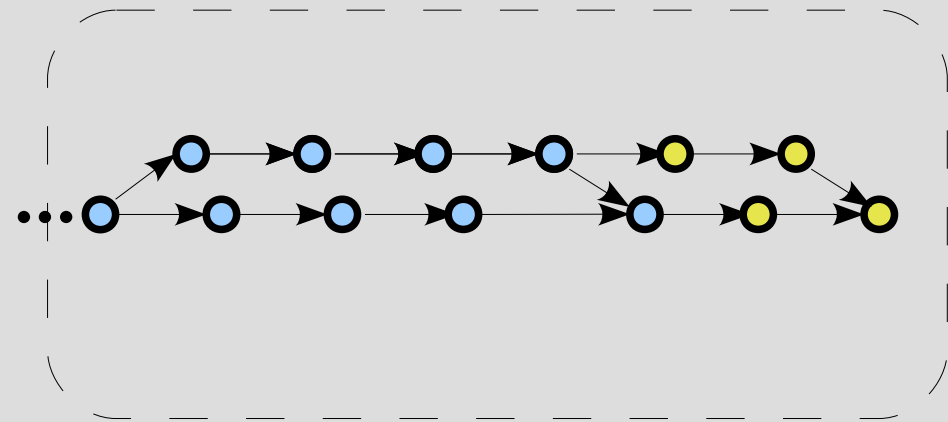
## Kernel hacker



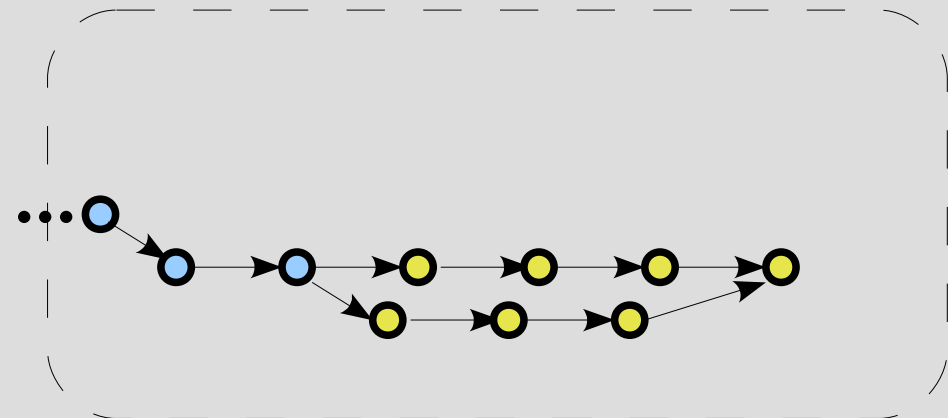
# Una sessione di un kernel hacker: sviluppo non lineare

- I due sviluppatori continuano il loro sviluppo in parallelo
- I commit sono creati dai due sviluppatori
  - Le patch usate per generare i commit sono spesso ricevute da collaboratori

## Maintainer



## Kernel hacker





# Una sessione di un kernel hacker: comunicazione del lavoro svolto

- Ciascuno dei commit è inviato anche alla mailing list del kernel
  - [linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org)
  - Punto di incontro pubblico fra maintainer, utenti, hacker
  - Non è il posto in cui vengono prese le decisioni strategiche (quello è il Linux Kernel Summit, aperto ai soli maintainer “che contano”)
  - Se il commit è interessante, si accende la discussione (e relativo feedback sul lavoro svolto)

# Una sessione di un kernel hacker: comunicazione del lavoro svolto

- La mailing list del kernel è usata anche per un altro motivo
  - Quando lo sviluppatore giudica le modifiche al proprio albero pronte per essere integrate nell'albero ufficiale, invia una richiesta di pull (**pull request**) a Torvalds
  - Torvalds viene notificato della possibilità di effettuare un pull (**git pull**) di un albero
  - Il “benevolent dictator” esamina il codice e...

# Lavata di capo...

On Sun, Sep 18, 2011 at 1:35 PM, Eric Dumazet <eric.dumazet@gmail.com> wrote:  
> [PATCH] tcp: fix build error if !CONFIG\_SYN\_COOKIE  
> commit 946cedccbd7387 (tcp: Change possible SYN flooding  
> messages) added a build error if CONFIG\_SYN\_COOKIE=n

Christ Eric, you clearly didn't even compile-test this one either.  
Which is pretty bad, considering that the whole and only *\*point\** of  
the patch is to make it compile.

The config option is CONFIG\_SYN\_COOKIES (with an 'S' at the end), but  
your patch has 'CONFIG\_SYN\_COOKIE' (without the S).

Which means that now it doesn't compile when syncookies are *\*enabled\**.  
I really wanted to release -rc7 today. But no way am I applying these  
kinds of totally untested patches. Can you guys please get your act  
Together?

PLEASE?

Stop with the "this might just work" crap. Because -rc7 is just too  
late to dick around like that.

Linus

# Ringraziamento

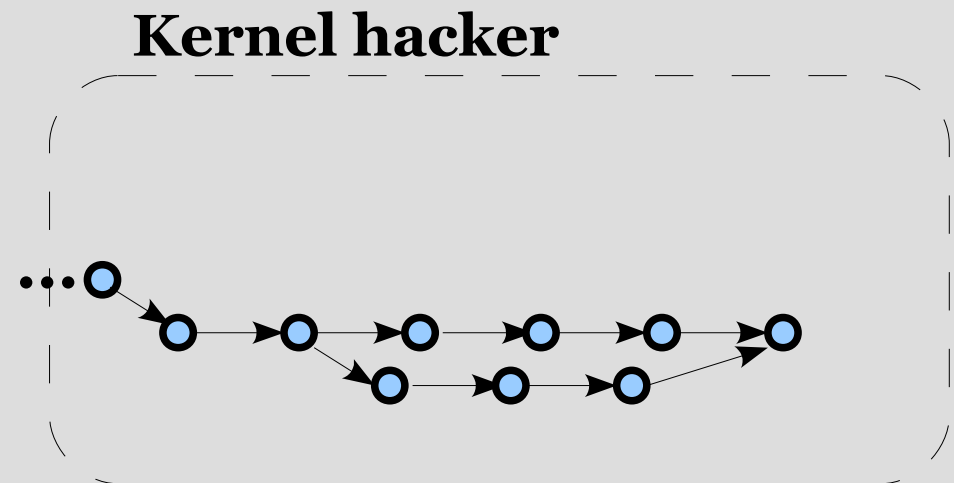
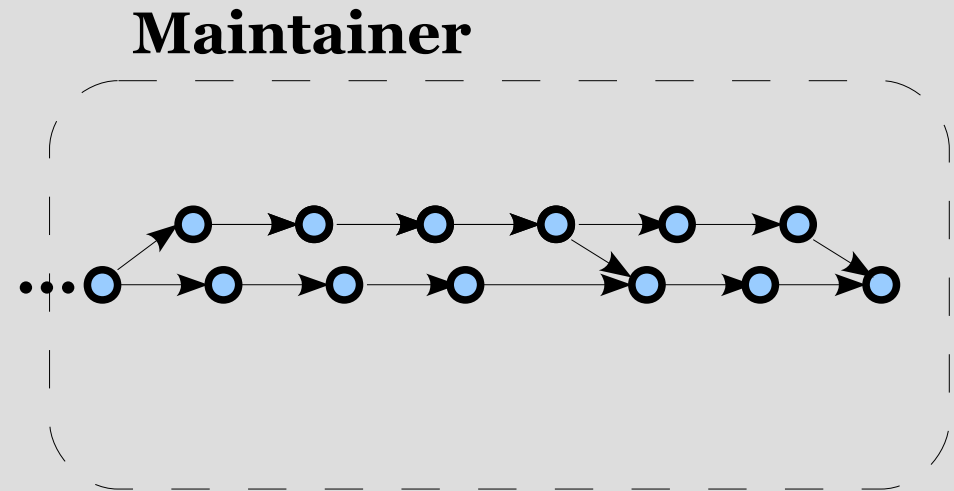
On Thu, Aug 25, 2011 at 1:21 PM, Arnaud Lacombe <lacombar@gmail.com> wrote:  
> On Thu, Aug 25, 2011 at 4:10 PM, Andy Lutomirski <luto@mit.edu> wrote:  
>>  
>> Arnaud, can you test this?  
>>  
> All good.  
>  
> Tested-by: Arnaud Lacombe <lacombar@gmail.com>

Thanks guys. Applied and pushed out,

Linus

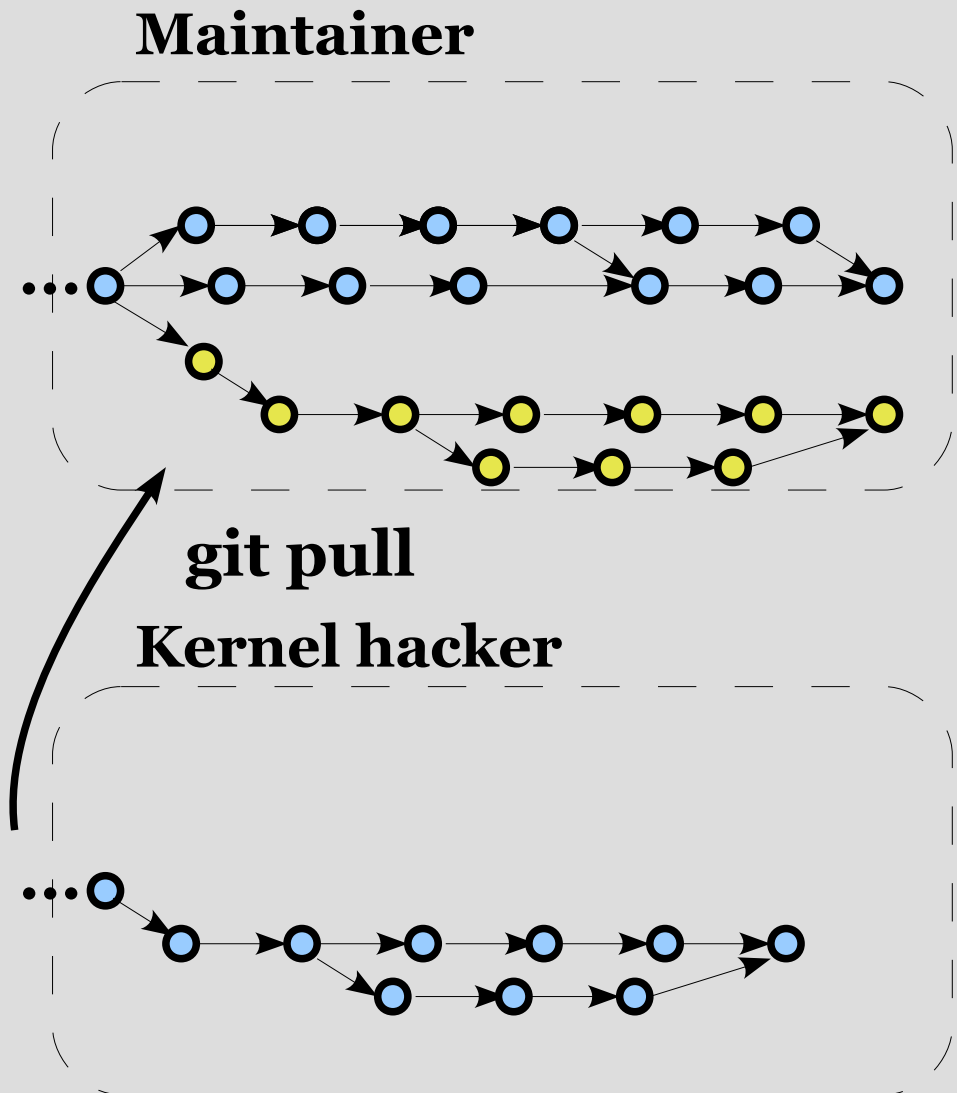
# Una sessione di un kernel hacker: sviluppo non lineare

- A questo punto, gli alberi dei due sviluppatori divergono
- È giunta l'ora di fondere le modifiche in un unico albero
  - Quello del maintainer principale



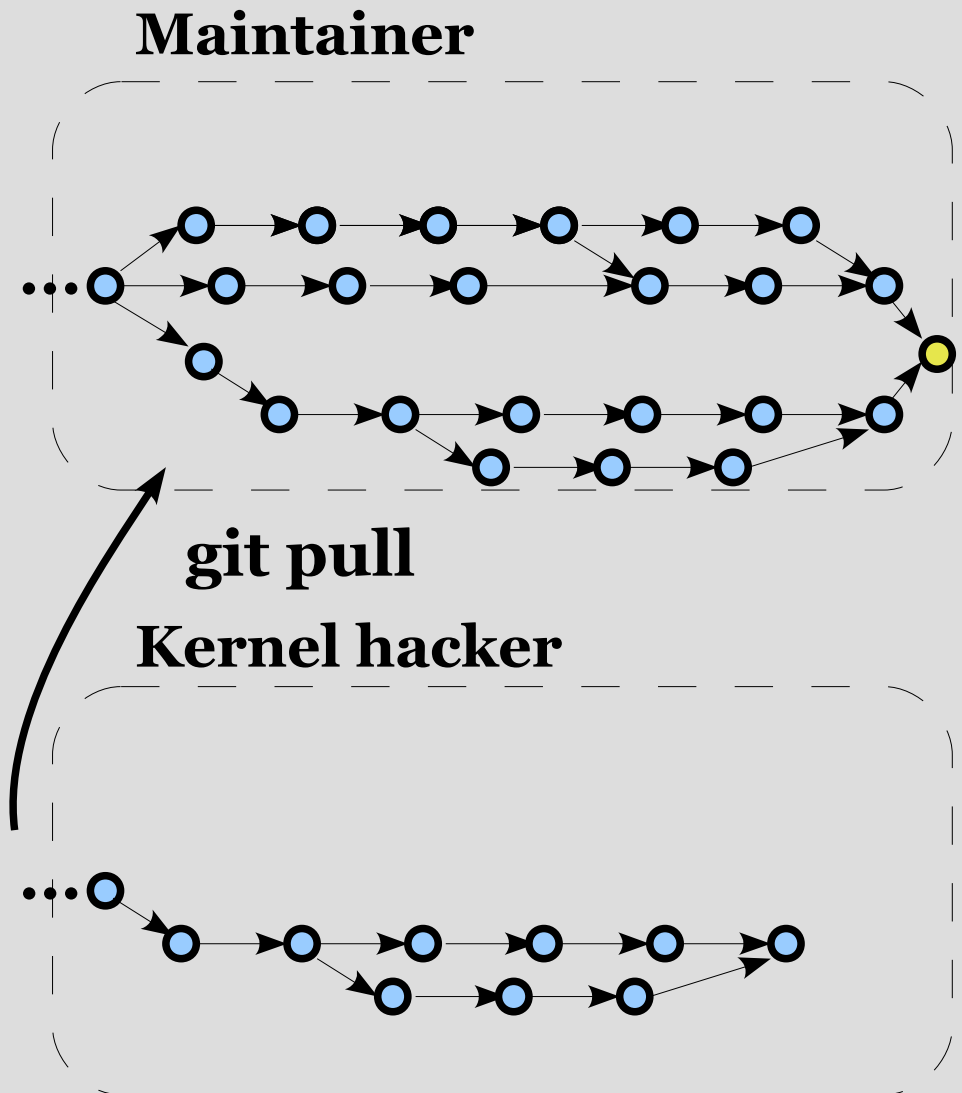
# Una sessione di un kernel hacker: sviluppo non lineare

- Il maintainer effettua il pull ed incorpora il branch del suo collaboratore in un proprio branch privato
- La prima funzione svolta dal pull è il **fetch** delle modifiche
  - Il maintainer si prende tutte le modifiche non presenti in locale



# Una sessione di un kernel hacker: sviluppo non lineare

- La seconda funzione svolta dal pull è il **merge** delle modifiche in master
  - Viene creato un nuovo commit che rappresenta il merge



# Una sessione di un kernel hacker: sviluppo non lineare

- Lo sviluppatore fa il pull a sua volta
  - Fetch
  - Merge
- I due repository sono finalmente sincronizzati
- Il ciclo si ripete “ad libitum”

