Version Control Systems

Ingegneria del Software T

Sinonimi

- Version Control
- Source Code Management (SCM)
- Source Control

→ Gestione dei codici sorgenti e delle versioni

Contesto

- Sviluppo distribuito (es. SourceForge, GitHub, Bitbucket, ecc.)
- Software House
- Singolo sviluppatore
- Studente…

In poche parole

Un Version Control System (VCS):

- Fornisce supporto alla memorizzazione dei codici sorgenti
- Fornisce uno storico di ciò che è stato fatto
- Può fornire un modo per lavorare in parallelo su diversi aspetti dell'applicazione in sviluppo
- Può fornire un modo per lavorare in parallelo senza intralciarsi a vicenda

→ Fornisce un modello di sviluppo

→ Aumenta la produttività

Best Practice

Usare un sistema di Version Control anche per lo sviluppo personale…

…a volte salva “la vita”.

Concetti di base (I)

- ***Repository***: un posto dove memorizzare i sorgenti
  - Tipicamente si trova su una macchina remota affidabile e sicura
  - Tutti gli sviluppatori condividono lo stesso repository
- ***Working folder***: cartella di lavoro
  - Ogni sviluppatore ne ha una collocata sulla propria macchina
  - Contiene una copia del codice sorgente relativo al progetto

Concetti di base (II)

- Sia il *Repository*, sia la *Working Folder* sono una gerarchia di cartelle o *directory*
- Il *workflow* di uno sviluppatore è tipicamente:
  - Copiare i contenuti del *repository* nella *working folder*
  - Modificare o aggiungere codice nella *working folder*
  - Aggiornare il *repository* incorporando i cambiamenti e le aggiunte
  - Ricominciare da capo…

Concetti di base (III)

Evitare di rompere il ciclo di sviluppo (***brake the tree*** – letteralmente rompere la gerarchia di cartelle)

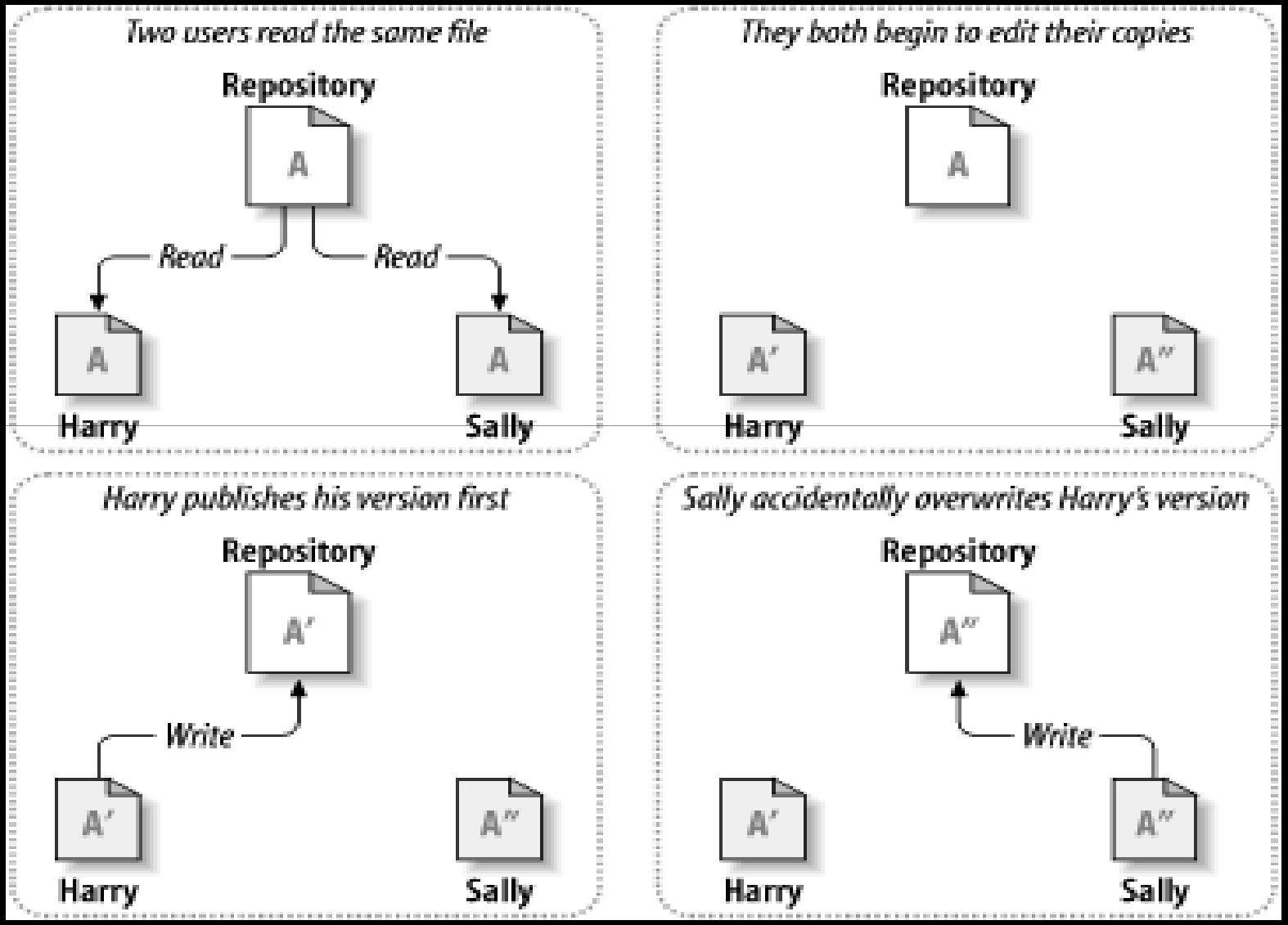
→ Il codice che viene memorizzato sul *repository centrale* deve portare il progetto in uno stato che consenta a chiunque (nel team) di proseguire nello sviluppo

→ Evitare di memorizzare codice che non compila o che non passa i test altrimenti tutto il team è costretto ad interrompere il ciclo di sviluppo

Repository = Time Machine

- Il *Repository* è un archivio di **ogni versione** di **ogni file** di codice sorgente
- Contiene la **storia** del progetto
- Rende possibile navigare indietro nel tempo e recuperare versioni vecchie dei file
  - Capire perché sono state fatte certe scelte
    - …e **chi** le ha fatte…
  - Capire perché sono stati introdotti nuovi bug
    - …e **chi** li ha introdotti…

Il problema da evitare



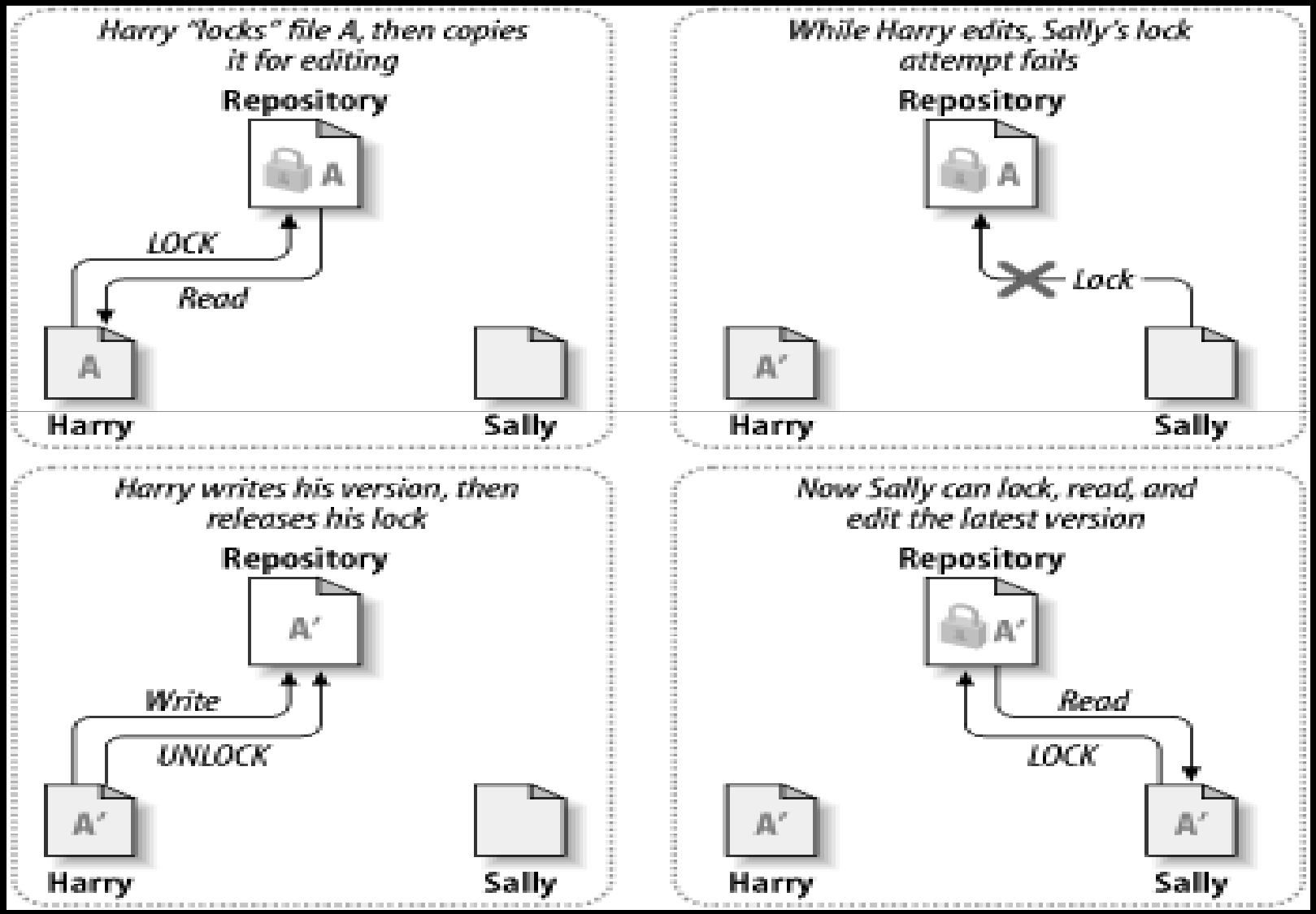
Modello *Lock-Modify-Unlock* (I)

- Tutti i file nella *working folder* sono inizialmente *read-only*
- L'utente A scarica o aggiorna un file sorgente nella sua *working folder*, lo rende *writable* e pone un *lock* su quel file nel *repository* → *checkout*
- L'utente A modifica la sua copia del file (che è *writable*)
- L'utente B non può a sua volta porre un *lock* sul file, al più può ottenerne una copia *read-only*
- L'utente A termina le modifiche sul file, salva il file sul *repository*, rende *read-only* il file nella *working folder* e toglie il *lock* → checkin
- L'utente B può ora porre un suo *lock* sul file per modificarlo

Modello *Lock-Modify-Unlock* (II)

- Uno sviluppatore deve effettuare un *checkout* prima di modificare un file → tutti sanno chi sta modificando cosa
- I *checkout* sono effettuati con *lock* esclusivi → solo uno sviluppatore alla volta può modificare un file

Modello *Lock-Modify-Unlock* (III)



Modello *Lock-Modify-Unlock* (IV)

Problemi:

- **Problemi di amministrazione**: un utente pone un *lock* poi se ne dimentica → ritardi e tempo perso
- **Serializzazione non necessaria**: un utente blocca un file e ne modifica una parte → impedisce ad un altro utente di modificare lo stesso file in una parte scorrelata dalla prima modifica
- **Falso senso di sicurezza**: un utente blocca un file e lo modifica, un altro utente blocca un altro file che dipende dal primo e lo modifica → le modifiche effettuate saranno compatibili?
- **Altri problemi**: come lavorare *offline*?

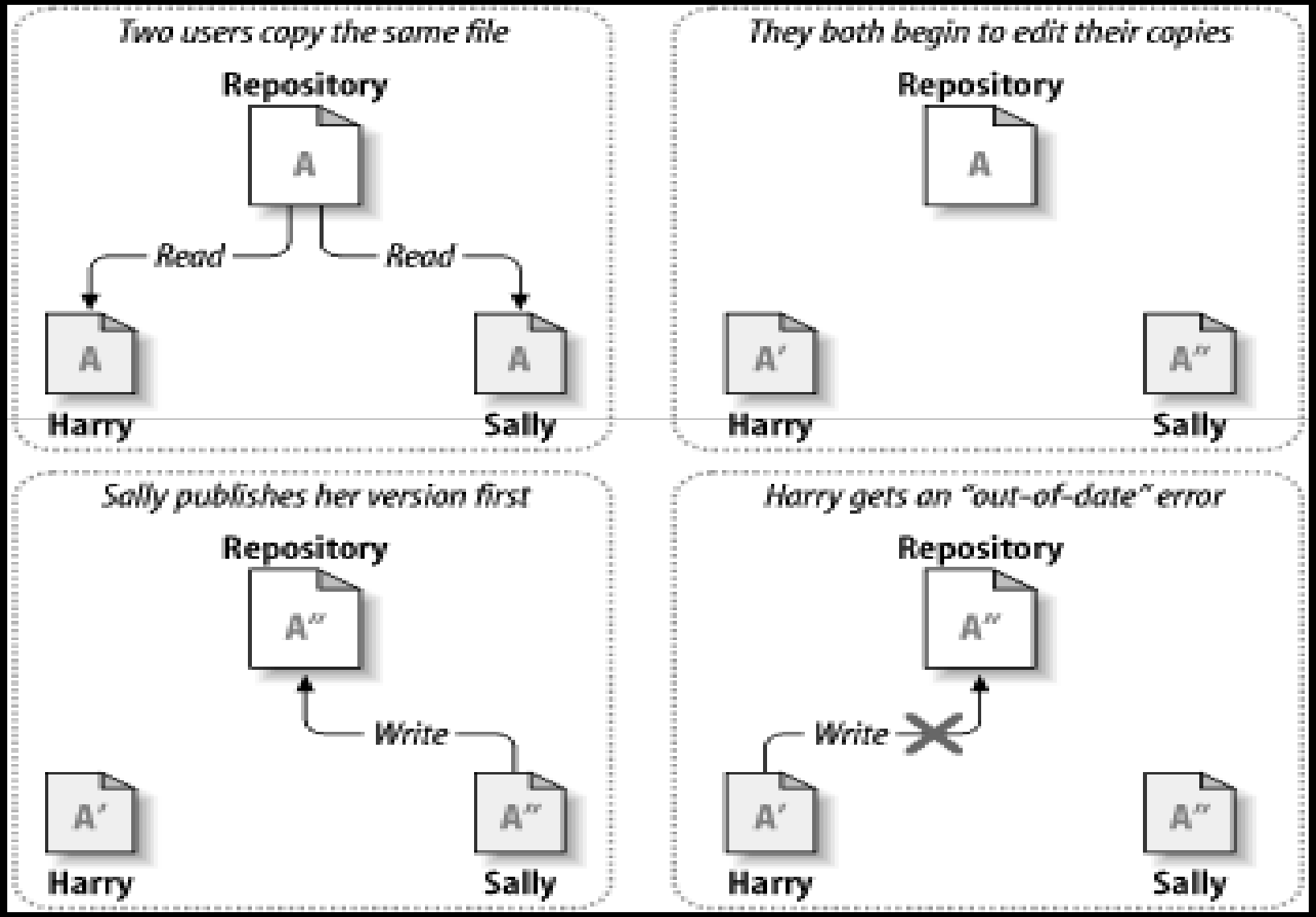
Modello *Copy-Modify-Merge* (I)

- **Non esistono *lock*** e tutti i file sono *writable*
- L'utente copia i file del *repository* nella sua *working folder*
- Modifica i file che desidera nella sua *working folder*
- Richiede un *update* della propria *working folder* (nel frattempo, qualcun altro potrebbe aver modificato il *repository*)
  - I file modificati vengono "fusi" (*merge*) con quelli contenuti nel *repository* nella versione finale del file
  - L'operazione di *merge* è tipicamente fatta in modo automatico e il risultato è tipicamente positivo: dopo un *merge* occorre verificare il corretto funzionamento dell'applicazione (*testing*…)
  - In alcuni casi il *merge* non può essere fatto automaticamente (*conflitti* – modifiche sulla stessa linea da parte di diversi sviluppatori); in questo caso i *conflitti* vengono *risolti* con strumenti che aiutano ad evidenziare le modifiche proprie e di altri
- Il salvataggio nel *repository* consente di pubblicare le modifiche effettuate (*commit*)

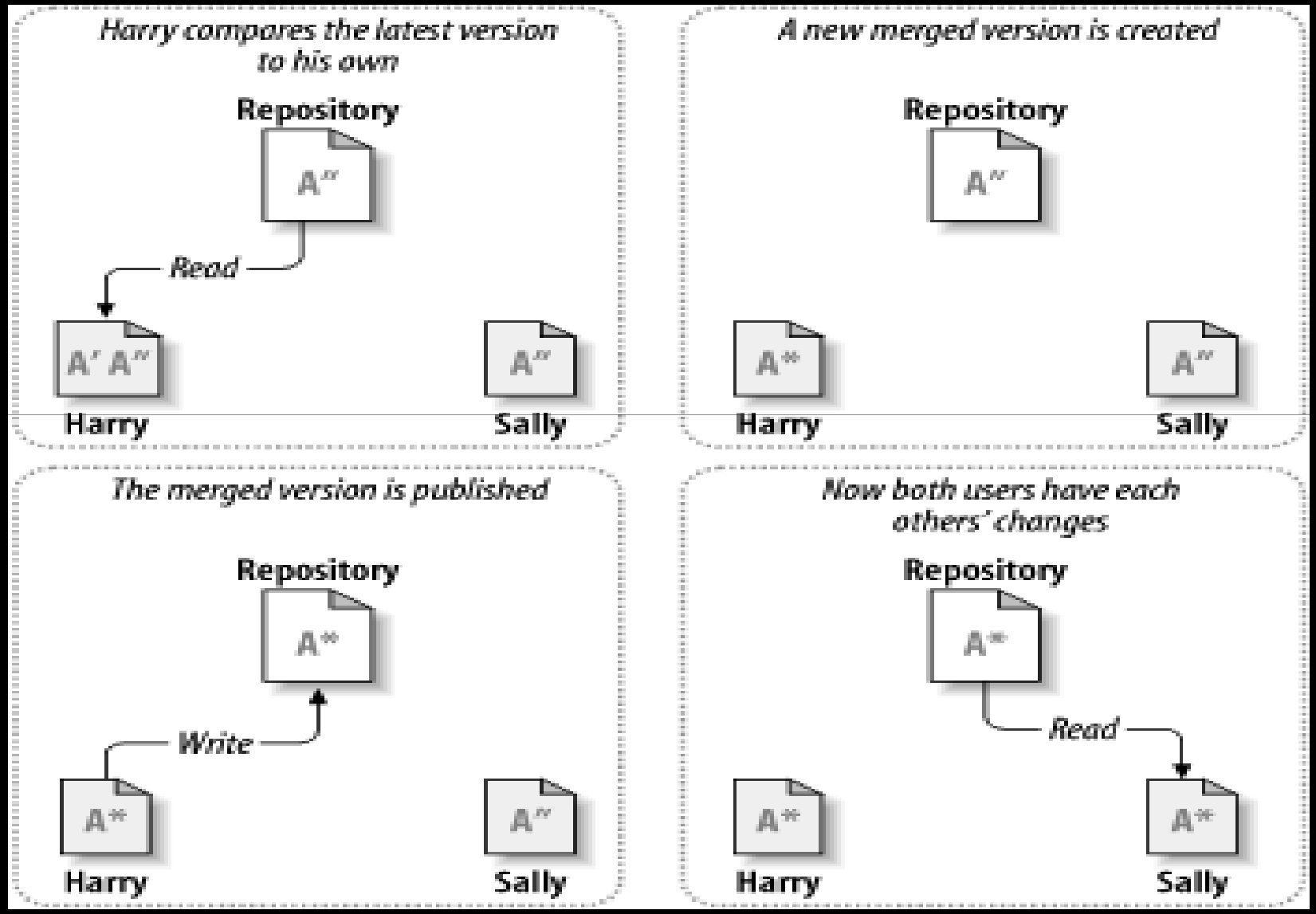
Modello *Copy-Modify-Merge* (II)

- Nessuno usa i *lock* quindi non si sa chi stia facendo cosa
- Il sistema si accorge se è necessario un *merge* oppure se è possibile salvare il file direttamente
- E' buona norma effettuare un *update* del contenuto della propria *working folder* prima di cominciare a lavorare
- Quando uno sviluppatore effettua un *commit* è sua responsabilità assicurarsi che i cambiamenti siano stati effettuati sull'ultima versione contenuta nel *repository* (v. slide precedente)

Modello *Copy-Modify-Merge* (III)



Modello *Copy-Modify-Merge* (IV)



Come funziona il merge?

- Tre file in gioco:
  1. File contenuto nel repository (ultima versione)
  2. File con modifiche dello sviluppatore
  3. File originale prima delle modifiche dello sviluppatore (in *working folder*)

- Se 1. e 3. sono uguali non serve il merge – si copia la nuova versione sul repository
- Si considera 3. come il “file originale”, 1. e 2. come file modificati rispetto all’originale

Quando viene rilevato un conflitto?

- Sui sorgenti quando:
  - Due sviluppatori modificano la stessa linea di codice; il primo fa un commit, il secondo un update… e rileva il conflitto.
- Sulla struttura dei file/direttori quando:
  - Due sviluppatori rinominano/spostano lo stesso file/direttorio.

Dopo un *merge* su un file…

- Dopo un *merge* il codice si compila sempre?
- Dipende: i cambiamenti di diversi sviluppatori, pur non entrando in conflitto secondo l'algoritmo di *merge*, possono generare inconsistenze finali
- Dopo il merge e prima di fare un commit, compilare e far girare la suite di test
  - La suite di test non esiste!? MALE!

La guerra dei mondi

- La dottrina "*lock-modify-unlock*" è più tradizionale e conservativa
- La dottrina "*copy-modify-merge*" è più rischiosa e libertaria
  - → Il rischio è che l'operazione di *merge* possa causare problemi e perdite di tempo
  - → L'accettazione del rischio (calcolato) consente di avere uno stile di sviluppo concorrente che permette agli sviluppatori di interagire molto meno

Quale scegliere?

- Il modello "*lock-modify-unlock*" è ormai preistoria…
- La tendenza è verso "*copy-modify-merge*" sempre più spinta
- Si perde anche l'idea della centralità del repository…
  - **Distributed Source Control Management Systems**

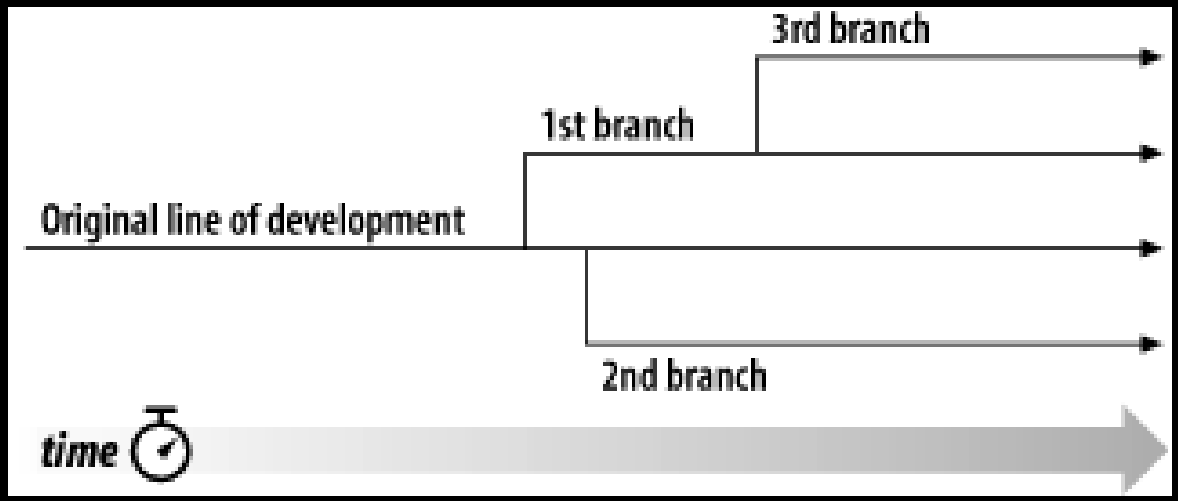
Trunk, Branch, Tag

Un modo ragionevole per organizzare un *repository* è fare in modo che contenga:

- Un tronco principale di sviluppo → *Trunk*
- Un luogo dove memorizzare le linee di sviluppo alternative → *Branch, Branches*
- Un luogo dove memorizzare le release stabili → *Tag, Tags*

Branch (I)

- Relativamente ad un progetto, un *branch* è una linea di sviluppo indipendente dalle altre
- Viene inizialmente generato come copia completa → condivide parte della storia



Branch (II)

- I *Branch*, ad esempio, consentono di iniziare lo sviluppo di una nuova *release* quando la precedente è ancora in fase di consolidamento
- Una volta terminato il consolidamento è possibile effettuare il *merge* fra il *branch* ed il *trunk*

  → I *bug* risolti in fase di consolidamento saranno fusi insieme alle modifiche apportate per incorporare le funzionalità della nuova *release*

Tag

- *Tag* = *Release*
- Viene memorizzato separatamente in modo da avere a portata di mano tutti i sorgenti relativi ad una certa *release*
- In questo modo non è necessario andare a ripescare dal *main trunk* i sorgenti andando indietro con le versioni dei file ← a quale versione di ogni file corrisponde una certa *release*?

Branch Merge

- Quanto può essere complicato fare il *merge* di un *branch*?
- Dipende dalle modifiche effettuate e dagli strumenti utilizzati:

  → Pochi file modificati, non modificata la struttura dei direttori: *merge* relativamente semplici

  → In alcuni casi si sceglie di ripartire da una versione precedente ed incorporare le modifiche in modo più opportuno

  → Troppo spesso è un vero bagno di sangue…

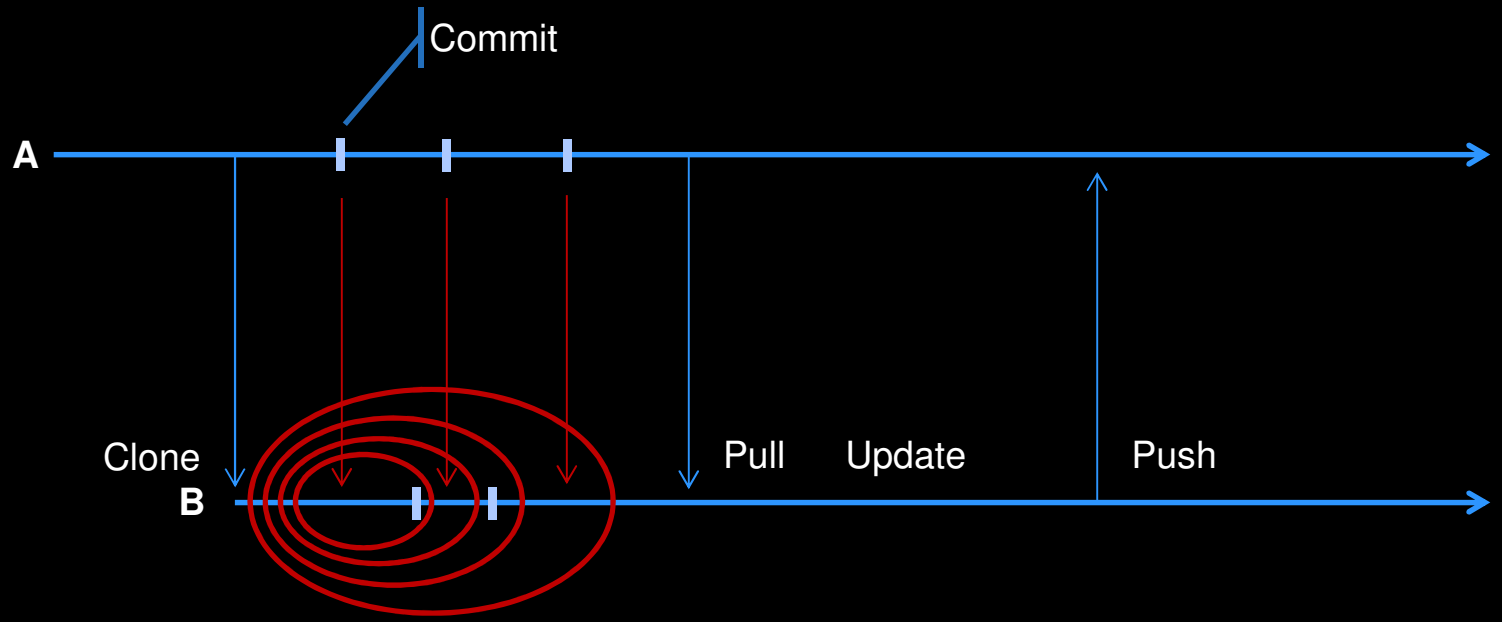
Distributed SCM

- **Il repository centralizzato non è più strettamente necessario**
  - I cambiamenti possono essere propagati da una working folder ad un'altra → **ogni working folder è anche un repository**
- Per creare la propria working folder:
  - Copia della working folder di un altro sviluppatore
  - Operazione di *clone* (equivalente) dal repo centralizzato (se esiste) o da una working folder di un altro sviluppatore
- La propria working folder è anche repository, quindi:
  - Le operazioni di *commit* e *update* sono operazioni *locali*
  - Per esportare i *commit* locali verso un altro repository/working folder, operazione di *push*
  - Per importare i *commit* effettuati su un altro repository/working folder, operazione di *pull* (dei change set)

Distributed SCM

- Ad ogni *commit* vengono memorizzati i cambiamenti sia nei sorgenti ma anche in termini di file rinominati, cartelle spostate, ecc.
  - Un *commit* crea un *change set*
- Un'operazione di *push* spedisce tutti gli «ultimi» cambiamenti al repository che riapplica tutti i cambiamenti effettuando il merge di un *change set* alla volta partendo dall'ultimo *pull* / *update*
- *Pull* e *update* sono operazioni effettuabili in istanti diversi:
  - *Pull*: recupera i *change set* dal repository
  - …si può continuare a lavorare…
  - *Update*: effettua il merge dei *change set* recuperati

Distributed SCM



- *Clone*: working folder/repository B creato da A
- *Commit*: operazione locale di memorizzazione del change set – ogni change set conosce il proprio parent
- *Pull*: vengono prelevati i change set dall'altro repository
- *Update/Merge*: scatena un'operazione di merge partendo dai change set più «vecchi» verso quelli più recenti
- *Push*: dopo aver effettuato il merge su B, pubblicazione di tutti i change set su A

Branching e Merging?

- Ogni working folder è un repository quindi, virtualmente, un *branch* indipendente
- La possibilità di effettuare molti commit quindi di avere molti change set «dettagliati», semplifica le operazioni di merge
- Il merge è un'operazione naturale e obbligatoria *pull/update*, *commit/push* sono operazioni di merge

Subversion

- SCM Open Source di tipo *Copy-Modify-Merge* **con repository centralizzato** e multipiattaforma
  - http://subversion.tigris.org/
- TortoiseSVN → Client grafico per Windows
  - http://tortoisesvn.tigris.org/
- RapidSVN → Client grafico multipiattaforma
  - http://rapidsvn.tigris.org/
- AnkhSVN → Plugin per Visual Studio
  - http://ankhsvn.open.collab.net/
- Subclipse → Plugin per Eclipse
  - http://subclipse.tigris.org

Mercurial

- Distributed SCM Open Source multipiattaforma
  - http://mercurial.selenic.com – installer sia di linea di comando, sia di tool grafico
- Visual HG → integrazione per Visual Studio
  - http://visualhg.codeplex.com/
- MercurialEclipse → Integrazione per Eclipse
  - http://www.javaforge.com/project/HGE

Git

- Distributed SCM Open Source nato su piattaforma Linux (recentemente portato anche su Win)
  - Inizialmente realizzato da Linus Torvalds per la gestione dei sorgenti del Linux kernel
  - Simile nell'utilizzo a Mercurial ma estremamente più potente… e pericoloso (l'history non è sacra…)
  - Prima è il caso di farsi le ossa con Mercurial

La scelta di Google

- Nel 2009, per Google Code, Google decise di fornire supporto ad un DVCS (oltre che al classico subversion) e si trattò di scegliere fra Mercurial e Git
- Il documento che giustifica la scelta fatta…
  - http://code.google.com/p/support/wiki/DVCSAnalysis
- Ha vinto Mercurial per la semplicità d'uso e per il supporto performante al protocollo HTTP
- Git continua ad essere molto popolare – Google ha recentemente dato supporto anche a Git