TECNOLÓGICA DEL ORIENTE
INSTITUCIÓN DE EDUCACIÓN SUPERIOR

GESTIÓN DEL SOFTWARE

RESOLUCIÓN DE CONFLICTOS

RESOLUCIÓN DE CONFLICTOS

Cuando varios desarrolladores trabajan sobre los mismos archivos de un proyecto, pueden surgir discrepancias entre los cambios realizados en distintas ramas. Este fenómeno, conocido como conflicto, exige una intervención manual para determinar qué fragmentos de código deben prevalecer. La resolución de conflictos implica revisar cuidadosamente el contenido en disputa, comprender el propósito de cada modificación y seleccionar la mejor combinación posible. Herramientas como Git o SVN ofrecen indicadores visuales y asistentes para facilitar esta tarea, pero la decisión final siempre recae en el criterio del equipo o del responsable del código. Así, la resolución de conflictos se convierte en una práctica esencial para preservar la coherencia y estabilidad del software en desarrollo (Chuck's Academy, s.f.).

¿Qué es un conflicto en control de versiones?

En el contexto del control de versiones, un conflicto representa una situación en la que dos o más cambios realizados sobre un mismo archivo no pueden integrarse automáticamente. Esto ocurre cuando múltiples colaboradores modifican las mismas líneas de código o estructuras en diferentes ramas o copias de trabajo, y el sistema de control no puede determinar de forma autónoma cuál modificación debe conservarse (Chacon & Straub, 2014).

Tanto en Git como en SVN, los conflictos son señales de que se requiere intervención humana para tomar una decisión informada sobre qué versión del contenido debe mantenerse, fusionarse o descartarse. Por lo general, los sistemas insertan delimitadores especiales en el archivo en disputa, para mostrar las partes en conflicto y permitir su edición manual.

Ejemplo en Git:

Supóngase que un desarrollador trabaja en una rama llamada feature-UI y modifica el archivo header.html agregando una nueva clase CSS. Al mismo tiempo, otro desarrollador hace una modificación diferente en el mismo archivo desde la rama main. Al intentar hacer un merge de feature-UI con main, Git detecta que ambas ramas editaron las mismas líneas y genera un conflicto.

Git marcará el archivo con indicadores como:

Figura 1. Ejemplo de conflicto

```
<<<<<<< HEAD
<div class="header blue-theme">
=======
<div class="header red-theme">
>>>>>>> feature-UI
```

Este marcador muestra claramente que existen dos versiones en disputa y que se debe elegir una o fusionarlas de forma adecuada.

Ejemplo en SVN:

En SVN, un conflicto puede surgir cuando un desarrollador actualiza su copia local (svn update) y se encuentra con un archivo que ha sido modificado por otros desde la última vez que se sincronizó. SVN generará automáticamente tres archivos: el archivo conflictivo, una versión base (.mine) y dos versiones de referencia (.rOLDREV y .rNEWREV) para ayudar al usuario a resolver el conflicto.

Identificación de conflictos

La identificación de conflictos en control de versiones es un proceso fundamental para garantizar la integridad del código fuente compartido por varios desarrolladores. Un conflicto se identifica cuando dos o más modificaciones concurrentes en un mismo archivo afectan exactamente la misma línea o estructura del código, y el sistema de control de versiones no puede resolverlo de forma automática. Reconocer estos conflictos de forma oportuna permite evitar errores en la integración y pérdidas de funcionalidad en el software (Chacon & Straub, 2014).

Tanto en Git como en SVN, existen mecanismos automáticos que alertan cuando ocurre un conflicto, generalmente durante una operación de merge (fusión), pull, o update. El sistema marca el archivo como "conflictivo" e impide continuar con el flujo de trabajo hasta que el problema sea resuelto manualmente.

Ejemplo en Git:

Cuando un desarrollador intenta fusionar la rama develop en main, y ambas ramas contienen cambios en la misma función del archivo login.js, Git lanza una advertencia:

Figura 2. Mensaje de conflicto en Git

```
Auto-merging login.js
CONFLICT (content): Merge conflict in login.js
Automatic merge failed; fix conflicts and then commit the result.
```

Además, el archivo login.js queda marcado con símbolos que indican las versiones en conflicto:

Figura 3. Conflicto en archivo login.js

```
<<<<<<< HEAD
function validateLogin() {
  // Versión actual
=======
function validateLogin(user) {
  // Nueva versión desde la rama develop
>>>>>>> develop
```

Este tipo de marcas le permite al desarrollador visualizar con claridad los bloques modificados, comparar ambos y decidir cómo integrarlos correctamente.

Ejemplo en SVN:

En SVN, cuando un usuario actualiza su copia local y encuentra cambios incompatibles con los suyos, el sistema lo notifica así:

Figura 4. Mensaje de conflicto en SVN

```
C    login.jsp
Updated to revision 123.
```

La "C" indica un archivo en conflicto, y el sistema genera tres archivos auxiliares:

- **login.jsp.mine:** versión local modificada.
- **login.jsp.r122:** versión base antes del conflicto.
- **login.jsp.r123:** nueva versión remota.

Estos archivos ayudan al desarrollador a comparar las diferencias y resolver el conflicto manualmente antes de confirmar los cambios.

Tipos de conflictos más comunes

En el contexto del control de versiones, especialmente al trabajar con sistemas distribuidos como Git o centralizados como SVN, es habitual enfrentarse a conflictos durante procesos de integración, fusión o actualización del código. Estos conflictos surgen cuando las herramientas de control no pueden determinar automáticamente cómo combinar cambios realizados por diferentes colaboradores. Identificar los tipos más frecuentes de conflictos permite a los equipos de desarrollo anticiparse y aplicar buenas prácticas para evitarlos o resolverlos con eficiencia (Chuck's Academy, s.f.).

1. Conflictos de contenido (content conflicts)

Este es el tipo más común. Ocurre cuando dos ramas modifican la misma línea de un archivo de forma diferente. El sistema de control de versiones no puede decidir cuál de las versiones debe conservar y requiere intervención manual.

Ejemplo (Git):

Figura 5. Conflicto de contenido en Git

```
<<<<<<< HEAD
return user.getEmail();
=======
return user.getCorreo();
>>>>>>> feature-internacionalizacion
```

El conflicto ocurre porque ambas ramas editaron la misma línea en Usuario.java con distintos nombres de función.

2. Conflictos de eliminación y modificación (delete/modify conflicts)

Se presentan cuando un desarrollador elimina un archivo que otro ha modificado en paralelo. El sistema no puede decidir si debe conservar la versión modificada o respetar la eliminación (Chuck's Academy, s.f.).

Ejemplo (SVN):

Un desarrollador elimina el archivo report.html en la rama limpieza, mientras otro realiza cambios en ese mismo archivo en la rama reporte-anual. Al hacer la fusión, SVN no sabe si debe borrar el archivo o mantenerlo con los cambios.

3. Conflictos de adición simultánea (add/add conflicts)

Suceden cuando dos ramas intentan crear un archivo con el mismo nombre y ruta, pero con contenidos distintos. Al fusionarlas, el sistema detecta que no puede mantener ambas versiones bajo el mismo identificador.

Ejemplo (Git):

Dos ramas independientes crean el archivo config.yml con configuraciones distintas. Al hacer el merge, Git no puede integrarlos automáticamente debido a la colisión de nombres.

4. Conflictos de cambio de nombre (rename conflicts)

Aparecen cuando un archivo se renombra en una rama, mientras que en otra se realizan cambios sobre el nombre original. Esto genera ambigüedad al momento de la fusión (Chuck's Academy, s.f.).

Ejemplo:

En la rama refactor, el archivo utils.js se renombra a helpers.js. Paralelamente, en main, otro desarrollador edita utils.js. Al fusionar, el sistema no sabe cómo reconciliar ambos cambios.

5. Conflictos binarios

Se generan cuando se modifican archivos binarios (como imágenes, PDFs, ejecutables) en paralelo. Como estos archivos no pueden fusionarse línea por línea, el sistema no puede resolver el conflicto automáticamente.

Ejemplo:

Dos diseñadores modifican banner.png desde ramas distintas. Al hacer merge, Git alerta sobre un conflicto binario y solicita una intervención manual para elegir qué versión conservar.

6. Conflictos por líneas en blanco o formato

Aunque menos críticos, los conflictos por formato ocurren cuando múltiples desarrolladores modifican espacios, tabulaciones o saltos de línea de forma distinta.

Herramientas como Git lo pueden marcar como conflicto si el archivo está configurado con reglas estrictas (Chacon & Straub, 2014).

Ejemplo:

Un desarrollador indenta un bloque con espacios y otro con tabulaciones en el mismo archivo y sección. Dependiendo de la configuración del repositorio, puede producirse un conflicto leve.

Herramientas para resolver conflictos

En el ámbito del desarrollo colaborativo de software, los conflictos durante la fusión de cambios son inevitables. Por ello, resulta esencial contar con herramientas especializadas que faciliten la detección, comprensión y resolución de estos conflictos. Tanto Git como SVN permiten gestionar estos casos manualmente desde la línea de comandos, pero diversas interfaces gráficas y asistentes visuales mejoran significativamente la experiencia del usuario, especialmente en proyectos de gran escala.

1. Git Merge Tool y Git GUI (Git)

Git incluye utilidades nativas como git mergetool, que permite abrir herramientas externas configuradas previamente para resolver conflictos. También ofrece Git GUI, una interfaz gráfica liviana para visualizar y resolver diferencias entre ramas (Chacon & Straub, 2014).

Ejemplo:

Un desarrollador realiza un merge entre dos ramas y se presenta un conflicto en main. py. Al ejecutar git mergetool, se abre automáticamente Meld (u otra herramienta configurada), mostrando lado a lado los cambios en conflicto y permitiendo al usuario seleccionar las partes que desea conservar.

```
git mergetool
```

2. TortoiseSVN (SVN)

TortoiseSVN es una interfaz gráfica ampliamente usada para trabajar con Subversion. Ofrece un visor de conflictos intuitivo que resalta las diferencias entre versiones y propone acciones para resolverlas (usar esta versión, la otra o una combinación de ambas).

Ejemplo:

Un colaborador actualiza su copia de trabajo y encuentra un conflicto en documentacion.txt. Al hacer clic derecho y seleccionar “Resolver conflictos”, TortoiseSVN abre un editor visual que permite comparar las versiones y elegir cómo integrarlas.

3. Visual Studio Code (VS Code)

VS Code detecta automáticamente conflictos en archivos de texto y los marca visualmente. Divide el contenido en secciones identificadas como “Incoming Change” y “Current Change”, lo cual permite resolver directamente desde el editor.

Ejemplo:

Durante una fusión, un archivo index.js presenta conflictos. VS Code muestra marcas como:

Figura 6. Conflicto en visual studio Code

```
<<<<<<< HEAD
console.log("Versión A");
=======
console.log("Versión B");
>>>>>>> rama-feature
```

El desarrollador puede seleccionar cuál bloque conservar o integrar ambos con una solución personalizada.

4. Meld

Meld es una herramienta gráfica de comparación y fusión de archivos y directorios. Es compatible tanto con Git como con SVN y ofrece una vista tridireccional que permite ver las versiones base, local y remota de un archivo en conflicto.

Ejemplo:

En un entorno Linux, al abrir meld archivo.java, el usuario puede comparar visualmente tres versiones del archivo y fusionar con solo unos clics.

5. Beyond Compare

Beyond Compare es una herramienta potente de pago (con versión de prueba) para la comparación y fusión de archivos. Su soporte para proyectos Git y SVN lo convierte en una opción popular entre equipos que valoran una visualización precisa de cambios y conflictos complejos.

Ejemplo:

Un equipo detecta un conflicto en un archivo de configuración extenso. Usando Beyond Compare, pueden analizar diferencias línea por línea, incluso en archivos CSV o XML, para tomar decisiones informadas.

6. KDiff3

KDiff3 es una utilidad gratuita que permite comparar hasta tres archivos simultáneamente. Resulta útil cuando los conflictos involucran múltiples ramas o versiones intermedias.

Ejemplo:

Durante la integración de una rama de larga duración, se requiere resolver un conflicto complejo entre la rama main, feature-a y una base común. KDiff3 muestra las tres versiones y permite realizar una fusión inteligente.

Tabla 1. Comparativa entre herramientas de control de versiones: Git y SVN

Característica	Git	SVN
Compatibilidad	Proyectos distribuidos	Proyectos centralizados
Interfaz	CLI (línea de comandos), GUI	CLI (línea de comandos), GUI
Sistema operativo	Windows, macOS, Linux	Windows, macOS, Linux
Nivel de experiencia	Intermedio - Avanzado	Principiante - Intermedio

Buenas prácticas para prevenir conflictos

En entornos colaborativos de desarrollo de software, prevenir conflictos durante la integración del código no solo ahorra tiempo, sino que también reduce errores y mejora la calidad del proyecto (Chuck's Academy, s.f.). Aunque los sistemas de control de versiones como Git y SVN están diseñados para gestionar estas situaciones, es preferible evitarlas desde la raíz aplicando un conjunto de buenas prácticas que promuevan la organización, la comunicación y la responsabilidad técnica dentro del equipo.

1. Mantener ramas actualizadas regularmente

Una práctica fundamental es integrar frecuentemente los cambios de la rama principal (como main o trunk) en las ramas de desarrollo activas. Esta acción permite detectar posibles conflictos de forma temprana y resolverlos mientras los cambios aún están frescos en la mente del desarrollador.

Ejemplo:

Si un desarrollador trabaja en una rama feature/login, debería realizar git pull origin main con regularidad para incorporar los cambios recientes y evitar que se acumulen diferencias importantes.

2. Evitar cambios simultáneos en las mismas líneas

Cuando varios desarrolladores trabajan en la misma parte de un archivo, las probabilidades de conflicto aumentan significativamente. Dividir las tareas de forma modular y asignar archivos o funciones específicas a cada integrante minimiza las colisiones (Chuck's Academy, s.f.).

Ejemplo:

En lugar de que dos desarrolladores editen el archivo usuarioController.js, uno puede encargarse de usuarioService.js y otro de usuarioValidator.js.

3. Hacer commits frecuentes y significativos

Registrar cambios pequeños y bien documentados facilita la identificación del origen de un conflicto si llegara a ocurrir. Además, permite que otros colaboradores comprendan fácilmente qué se modificó y por qué.

Ejemplo:

En vez de un único commit con el mensaje “Cambios varios”, es preferible realizar varios commits cómo “Agregar validación de correo electrónico” o “Refactoriza función de autenticación”.

4. Usar ramas para tareas específicas

Gestionar el trabajo mediante ramas dedicadas por funcionalidad, corrección de errores o mejoras permite mantener un historial ordenado y simplifica el proceso de revisión y fusión. Evitar desarrollar directamente sobre la rama principal ayuda a preservar la estabilidad del código base (Chacon & Straub, 2014).

Ejemplo:

Para una nueva funcionalidad de reportes, se puede crear una rama feature/reportes-exportación en lugar de trabajar sobre main.

5. Revisar código antes de fusionar (Pull Requests / Merge Requests)

Los sistemas colaborativos como GitHub, GitLab o Bitbucket permiten establecer flujos de revisión donde otro miembro del equipo inspecciona los cambios antes de integrarlos al repositorio. Esta práctica ayuda a detectar posibles conflictos o errores lógicos con antelación.

6. Establecer convenciones de codificación

Cuando todos los miembros del equipo siguen el mismo estilo de código (indentación, nombres de variables, estructura de archivos), se reduce la probabilidad de conflictos provocados por cambios meramente estéticos o de formato.

Ejemplo:

Definir que todas las funciones deben escribirse en notación camelCase y que los archivos deben finalizar con una línea en blanco.

7. Comunicación constante entre desarrolladores

El diálogo abierto entre los miembros del equipo permite evitar escenarios en los que dos personas trabajan sobre el mismo módulo sin saberlo. Herramientas como Slack, Discord o Microsoft Teams pueden integrarse al flujo de trabajo para facilitar esta sincronización.

Confirmación y finalización después de resolver conflictos

Una vez resuelto un conflicto en un sistema de control de versiones como Git o SVN, el desarrollador debe llevar a cabo una serie de pasos esenciales para confirmar los cambios y concluir el proceso de integración de forma ordenada y segura. Esta etapa es crítica, ya que garantiza que el código resuelto se guarde correctamente, quede registrado en el historial del proyecto y esté listo para ser fusionado sin errores (Chacon & Straub, 2014).

1. Verificación del estado del repositorio

Después de editar y corregir los archivos en conflicto, se recomienda revisar el estado del repositorio para asegurarse de que no queden conflictos pendientes. En Git, por ejemplo, el comando:

git status

muestra si hay archivos marcados como unmerged, conflict fixed, o listos para agregar al área de preparación (staging area).

2. Añadir los archivos corregidos

Una vez verificado que los conflictos fueron resueltos adecuadamente, el siguiente paso es añadir los archivos modificados al área de preparación para confirmarlos. En Git, esto se hace con:

git add nombre-del-archivo
o si se desea incluir todos los archivos resueltos:
git add .

Esto indica al sistema que los conflictos han sido solucionados y que los archivos pueden ser incorporados en el próximo commit.

3. Confirmar (commit) la resolución

Luego de añadir los archivos, se realiza la confirmación para registrar los cambios. Git solicitará un mensaje de confirmación, que idealmente debe indicar que se trata de la resolución de un conflicto (Chacon & Straub, 2014).

Ejemplo de confirmación:

git commit -m "Resuelve conflicto entre main y feature/login"

Este mensaje debe ser claro y específico para que otros colaboradores puedan entender el contexto de la fusión sin necesidad de revisar todo el historial.

4. Finalización del proceso de fusión

Si la resolución de conflictos forma parte de una fusión (merge), al realizar el commit final, Git completa automáticamente la operación de merge. Esto se refleja en el historial del repositorio como un nodo de fusión que une las dos ramas involucradas.

En el caso de SVN, tras modificar los archivos en conflicto, se marcan como resueltos usando:

svn resolved nombre-del-archivo

y luego se realiza un commit para guardar los cambios:

svn commit -m "Conflicto resuelto entre trunk y rama de reporte"

5. Pruebas y validación posterior a la fusión

Antes de continuar con otros desarrollos, es recomendable compilar, ejecutar y probar el proyecto para verificar que los cambios no han introducido errores. A veces, un conflicto mal resuelto puede pasar desapercibido y causar problemas en el tiempo de ejecución.

BIBLIOGRAFÍA

- Cabrera, E. (2015). Flujo de trabajo en SVN [Imagen]. https://eudriscabrera.com/images/blog/2015/flujo-svn-300x205.png
- Chacon, S., & Straub, B. (2014). Pro Git (2.ª ed.). Apress. https://git-scm.com/book/es/v2
- Chuck's Academy. (s.f.). Resolución de conflictos en Git. https://www.chucksacademy.com/es/topic/git-conflicts
- Guillamón Morales, A. (2013). Manual desarrollo de elementos software para gestión de sistemas. Editorial CEP, S.L. https://elibro.net/es/lc/tecnologicadeloriente/titulos/50603
- Pérez Martínez, E. (2015). Desarrollo de aplicaciones mediante el Framework de Spring. RA-MA Editorial. https://elibro.net/es/lc/tecnologicadeloriente/titulos/107207
- Villacis, G. (2024, julio 17). Tutorial de flujo de trabajo con Git: ¡Empieza a usar los comandos básicos de Git AHORA!. DEV Community. https://dev.to/villacisg93/tutorial-de-flujo-de-trabajo-con-git-empieza-a-usar-los-comandos-basicos-de-git-4dbi