II CONPESQ
Congresso de Pesquisa, Pós-Graduação e Inovação
Os novos rumos da ciência pós-pandemia
12 a 16 de abril de 2021 Universidade Federal do Cariri - UFCA

É possível recuperar cenários de merge que não aparecem no grafo de commits do Git?

Pedro Henrique Lopes dos Santos, UFCA,
lopes.pedro@aluno.ufca.edu.br

Paola Rodrigues de Godoy Accioly, UFCA,
paola.accioly@ufca.edu.br

RESUMO: Ao existir um grupo de desenvolvimento de sistemas trabalhando em um mesmo repositório de código-fonte, é bem provável que existirão erros e a medida em que o tamanho dele aumente e a cada integração de código, conflitos podem ocorrer. Pesquisadores têm estudado de várias maneiras esses conflitos, seja prevenindo-os ou buscando maneiras de resolver parte deles automaticamente. No entanto, nenhum dos estudos prévios leva em consideração dados não rastreáveis através do histórico de commits de repositórios que usam Git como sistema de controle de versão. Esses dados não são rastreáveis porque desenvolvedores podem usar comandos como rebase, cherry-pick e squash que reescrevem a história do repositório, apagando a evidência de integrações de código e seus conflitos. Nosso objetivo nesse estudo e avaliar se é possível e viável recuperar dados sobre essas integrações, já que, independentemente do método de merge usado, os conflitos estão ocorrendo e sendo resolvidos localmente. Além de que, não documentar esses casos pode acarretar em um não entendimento sobre o repositório, já que essas ações locais podem ter consequências em algum conflito que possa ocorrer futuramente. A análise indicou e nos mostrou que alguns dados realmente são perdidos e não conseguimos recuperar, ao contrário de outros que são casos mais básicos de se trabalhar, como por exemplo um “rebase fast-forward”.

PALAVRAS-CHAVE: Sistema de Controle de Versão, Conflitos de Integração de Código, Merge, Rebase, Cherry-Pick, Squash.

ABSTRACT: When there is a group of system development working in the same source code repository, it is very likely that there will be errors and as its size increases and with each code integration, conflicts can occur. Researchers have studied these conflicts in various ways, either by preventing them or looking for ways to resolve part of them automatically. However, none of the previous studies takes into account non-traceable data through the history of commits from repositories that use Git as a version control system. This data is not traceable because developers can use commands such as rebase, cherry-pick and squash that rewrite the repository's history, erasing evidence of code integrations and their conflicts. Our objective in this study is to assess whether it is possible and feasible to recover data on these integrations, since, regardless of the merge method used, conflicts are occurring and being resolved locally. Besides, not documenting these cases can lead to a lack of understanding about the repository, as these local actions can have

consequences in some conflict that may occur in the future. The analysis indicated and showed us that some data are actually lost and we were unable to recover, unlike others that are more basic cases of work, such as a “fast-forward rebase”.

Keywords: Version System Control, Code Integration Conflicts, Merge, Rebase, Cherry-Pick, Squash.

1 INTRODUÇÃO

Em projetos maiores, para o desenvolvimento de certa ferramenta, é ideal que o trabalho seja dividido em partes, e posteriormente, o trabalho de várias pessoas em cada uma delas. Uma das ferramentas mais utilizadas é o Git, plataforma estudada por esse trabalho.

Existem comandos no Git que integram o código e ainda por cima ocultam esse merge do histórico do repositório, como por exemplo: Cherry-Pick, Rebase e Squash. Por isso, é interessante, academicamente falando, estudá-los e documentar os resultados, já que não sabemos com que frequência esses tipos de integração ocorrem, podendo resultar em um problema maior do que o imaginado, podendo ser analogamente descritos como a ponta superficial de um enorme iceberg.

Assim, o objetivo deste trabalho é estudar repositórios privados dos desenvolvedores a fim de entender, documentar e até recuperar os cenários de integrações feitos pelos comandos citados.

2 INTEGRAÇÃO DE TAREFAS DE DESENVOLVIMENTO USANDO O GIT

No Git, quando um desenvolvedor termina suas atividades, é necessário um processo de integração, que sincroniza os dados novos com o desenvolvimento do que já tínhamos, considerando que enquanto esse programador estava trabalhando, o repositório remoto tenha evoluído.

Para realizar a integração das tarefas de desenvolvimento, o Git oferece várias opções de comandos. Entre elas, a mais intuitiva é o comando “git merge”. Quando o desenvolvedor opta por esse comando, o cenário de integração fica salvo no histórico de desenvolvimento.

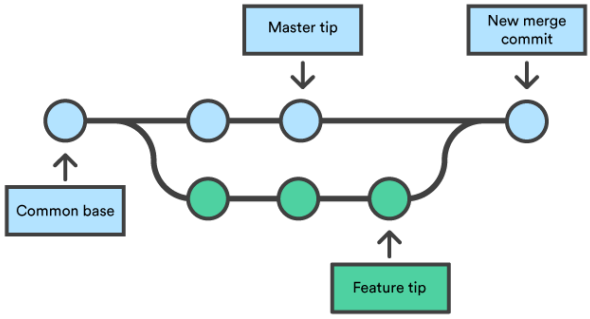
No entanto, existem outras maneiras de realizar essas integrações sem salvá-las no histórico: usando rebases, squashs, cherry-picks e stash apply. Nesta seção iremos descrever melhor cada um desses comandos.

2.1. MERGE

O merge é a maneira de unir linhas de desenvolvimento independentes, integrando-as em um único branch. Nos cenários nos quais o merge é usado para mesclar dois branches, o git pega dois indicadores de commit, que geralmente são as pontas do branch e encontra uma base em comum para os dois. Achado a base, são combinadas as alterações de cada sequência de commits.

Este comando não deixa o histórico de desenvolvimento linear, já que o commit resultante do merge aponta para dois pais.

Figura 1 – Cenário de Merge



Fonte: Git Merge. https://www.atlassian.com/br/git/tutorials/using-branches/git-merge. Acesso em: 06/05/2020.

A Figura acima mostra que o “New merge commit” é o resultado do merge entre o branch “Feature” e o “Master”, e “Common base” é a base em comum dos commits que sofreram o merge.

Nesse caso, identificar os cenários de integração que ocorreram durante o desenvolvimento de um projeto se torna trivial, pois basta coletar os commits que possuem mais de um pai.

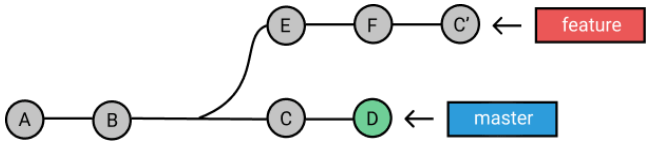
2.2. CHERRY-PICK

É um comando que permite ao usuário escolher commits que deseja em um determinado branch. É útil porque permite a adição de apenas commits importantes, evitando aqueles que estão em teste ou até que possuem conflitos.

Como sabemos, cada commit possui uma identificação única. Ao utilizarmos o comando “git cherry-pick <id do commit>”, o git pega o conteúdo desse commit e o fixa no

lugar onde estamos no branch atual e salva isso com um id diferente.

Figura 2 - Cenário de Cherry Pick



Fonte: Cherry Picking with git. https://dev.to/neshaz/cherry-picking-with-git-4pmj. Acessado em: 06/05/2020.

Na Figura, por exemplo, o usuário, deu um “git checkout feature”, para ir ao branch no qual ele quer que o commit final esteja e então um “git cherry-pick C”. O resultado é o commit C’.

2.3. STASH APPLY

Outra forma de integrar, é utilizando a sequência de comandos “git stash”, “git pull” e “git stash apply”. O stash permite com que o usuário salve suas modificações ainda não commitadas em uma espécie de backup, para caso o desenvolvedor precise utilizá-las no futuro, também deixando a árvore de trabalho limpa.

Ao utilizar o pull, o conteúdo do repositório local é atualizado, e então, ao dar um stash apply, o conteúdo da pilha de backup é recuperado e são aplicadas as mudanças.

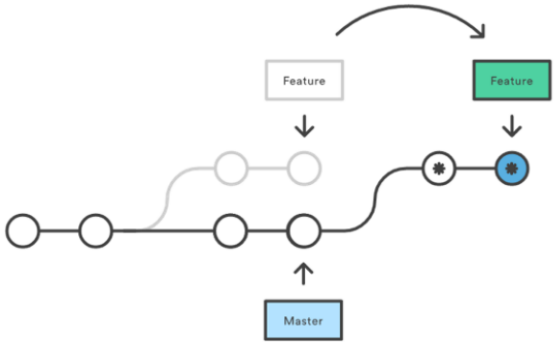
2.4. REBASE

Rebase, basicamente, é o ato de mover os commits de um branch para outro. A maneira que o Git utiliza para fazer isso é aplicando os commits de um por um, em ordem, na ponta do branch que foi especificado no comando.

Uma das coisas que é interessante de se mencionar em relação aos rebases, é um dos motivos para ele ser realizado: Manter o histórico da ramificação “limpo”, já que o grafo se torna linear.

O que ele faz, resumidamente, é reproduzir commit por commit as ações que o desenvolvedor desejar.

Figura 3 - Cenário de Rebase



Fonte: Git Rebase. https://www.atlassian.com/br/git/tutorials/rewriting-history/git-rebase. Acessado em: 07/05/2020.

2.4.1. Rebase Interativo e Squash

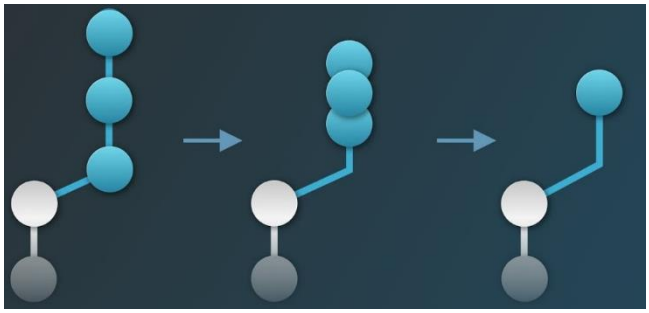
O Squash nada mais é do que um dos possíveis comandos ao se fazer um rebase interativo. Para fazer de um rebase interativo, basta adicionar o argumento "-i" ou "--interactive", assim, se abrirá uma espécie de menu com todos os comandos possíveis de serem executados, com todos os commits com o comando "pick" por padrão.

Ao se executar um rebase interativo, podemos utilizar seis comandos: pick, reword, edit, squash, fixup e exec.

O primeiro, só significa que o commit está incluso no rebase, e por isso ele vem por padrão em todos os commits que sofrerão o comando. O segundo pode ser considerado um pick, mas com a função de reescrever a mensagem do commit. O edit nos permite adicionar ou alterar o commit por inteiro, nos permitindo, por consequência, dividir um commit grande em commits menores ou remover o que quisermos.

O squash nos permite reescrever o grafo de commits, condensando várias alterações (commits) em um só. Então, ao se incluir um merge nos commits a sofrer o squash, perdemos qualquer informação que tínhamos e sequer temos alguma evidência de que aquela integração aconteceu.

Figura 4 - Cenário de Squash



Fonte: Combining git Commits with Squash.
https://www.youtube.com/watch?v=V5KrD7CmO4o. Acesso em: 07/05/2020.

No caso da figura, os três commits em azul sofreram um squash e se tornaram um só.

Continuando, o fixup, semelhante ao squash, faz um merge com o commit acima dele, mas o que sofreu o merge tem sua descrição apagada e a mensagem do commit anterior é mantida. E por último, o exec, nos permite executar comandos shell em um commit. [1]

3 É POSSÍVEL RECUPERAR CENÁRIOS DE INTEGRAÇÃO UTILIZANDO LOGS DOS REPOSITÓRIOS PRIVADOS DOS DESENVOLVEDORES?

Com exceção do Merge, todos os métodos citados na seção anterior reescrevem o histórico e podem ocultar cenários de integração e, consequentemente, conflitos.

Em repositórios, conflitos estão ocorrendo e sendo resolvidos em âmbito local. E sendo nosso objetivo com este artigo analisar se é possível, utilizando os arquivos de logs do desenvolvedor, recuperar esses cenários.

3.1 ANÁLISE COM EXEMPLOS TOY

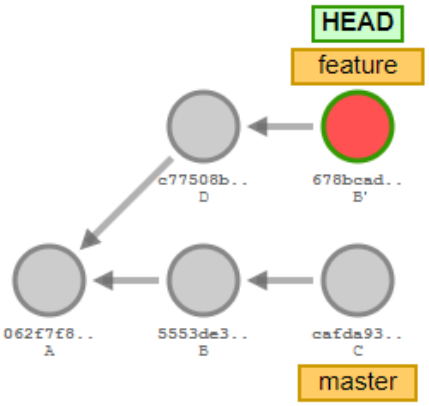
Aqui, os estudos foram feitos com exemplos toy criados em repositórios locais e exclusivamente feitos para entender como os comandos executados afetam o arquivo de log do Git.

3.1.1 Cherry-Pick

Ao se fazer um cherry-pick, localmente, o que temos de informação no log é

que este comando aconteceu e o nome da nova mensagem do commit, sem nenhuma informação a mais sobre o hash antigo a não ser que o desenvolvedor utilize o parâmetro "-x".

Figura 5 – Cenário toy de Cherry-Pick



Na figura, temos um cenário genérico, na qual temos o branch feature, cuja base é A, e nele, temos B', que é o resultado de um cherry-pick no commit B. No log, isso se reflete assim:

Figura 6 – Log do Cenário toy de Cherry-Pick

```
commit (initial): A
commit: B
commit: C
checkout: moving from master to feature
commit: D
checkout: moving from feature to master
checkout: moving from master to feature
commit (cherry-pick): B'
```

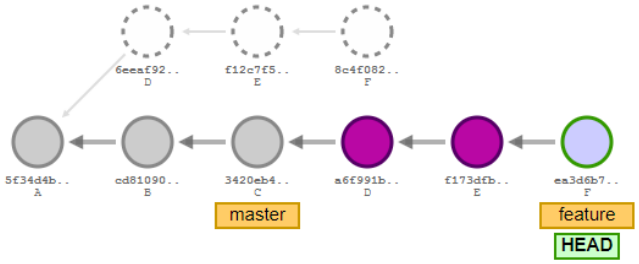
Acima, podemos ver como o cherry-pick aparece no log e no caso, podemos perceber que houve conflito ao dar o comando, levando em consideração a necessidade de se ter feito um commit.

Enfim, sobre os cherry-picks, como uma tentativa de recuperar os cenários de integração, tentamos analisar a forma, o tamanho da tree do commit, além do nome dos blobs, mas nenhuma dessas análises foram conclusivas e todas são situacionais, ou seja, variam de cenário dependendo da situação, se tornando inviável, computacionalmente falando, de perceber um padrão e elaborar um script com isso.

3.1.2 Rebase e Squash

Nos rebases, o que temos é que no log, quase todas as informações são guardadas, desde um simples checkout até o abort, na qual toda a operação é cancelada.

Figura 7 – Cenário toy de Rebase



Neste caso, tínhamos o branch feature, que continha os commits D, E e F, e o master, que tinha B e C, ambos com base em A. Então, foi feito um rebase e todos os commits do feature foram reaplicados na ponta do master.

A mesma coisa acontece com o rebase interativo, nos logs, são exibidos como mensagens seguidas, de acordo com o que o desenvolvedor tiver feito.

Figura 8 – Exemplo de log onde foi feito um Squash

```
commit: F
rebase (start): checkout master
rebase (continue): squash commit
rebase (squash): # This is a combination of 2 commits.
rebase (squash): squash commit
rebase (finish): returning to refs/heads/feature
```

Neste caso, temos um exemplo de squash. O que podemos perceber logo de cara, é que ocorreu conflito, pelo “continue” entre parênteses. Todas as mensagens ficam como seguidas no log.

Considerando um bloco de rebase não interativo, existem padrões que se repetem, coisa que nos permitiu automatizar o processo de recuperação dos cenários criando um script.

Assim, resumidamente, o script olha para a primeira linha do bloco dos rebases e pega os hashes para então como último passo, executar um merge-base entre os dois pais que achamos, para termos o cenário de integração completo composto pelos 3 commits.

3.2 ANÁLISE COM EXEMPLOS TOY

Então, como uma maneira de verificar se podemos automatizar a recuperação dessas informações, foi desenvolvido um script na linguagem Java, que utilizamos para analisar arquivos de logs reais que foram coletados previamente, durante a fase de estudo do trabalho de graduação da Marcela Bandeira Cunha [2].

Este script funciona para Rebases, já que com as outras formas de integração, não conseguimos recuperar as informações usando apenas o log como entrada.

Para cada arquivo do histórico de um desenvolvedor que tínhamos, o procedimento feito foi, clonar o repositório e então, colar a pasta “logs” que os desenvolvedores nos cederam a fim de conseguir recuperar as bases com um “git merge-base” e então, executar o script no repositório.

Assim, ao fazer o procedimento do parágrafo anterior nos repositórios, recebemos uma planilha como saída, na qual nos são mostradas quatro colunas. A primeira é a base, recuperada pelo merge-base do hash que temos na segunda e na quarta coluna, que são as pontas dos branches existentes e por fim, a terceira coluna é o hash do commit da ponta do branch em que o comando foi executado.

Um exemplo da tabela descrita no parágrafo anterior pode ser encontrado nas referências [3].

Segundamente, reutilizando, adaptando e expandindo o software, agora conseguimos utilizar a API do GitHub para analisarmos o histórico do repositório olhando o pai de um commit recursivamente, conseguindo recuperar mais rebases, já que eliminamos o problema da perda da referência local que tínhamos antes.

Recursivamente, foi estabelecido de que no grafo, o software iria olhar em uma profundidade de até 30 commits procurando uma base.

Como saída, além do cenário de integração que tínhamos antes, agora temos outras informações quantitativas como o número de rebases do repositório, se foi um fast-forward, o número de bases que puderam ser encontradas, o número de vezes em que o limite de profundidade foi atingido e o número de vezes nas quais não se achou a referência desse commit. Ao finalizar, o script contabiliza e condensa tudo em uma tabela.

Um exemplo de tabela resultante pode ser encontrada nas referências [4].

4 CONCLUSÃO E TRABALHOS FUTUROS

Com relação aos cherry-picks, não podemos recuperar os cenários por falta de informações e pela aleatoriedade de algumas variáveis nos impediu de elaborar um script para isso.

Enfim, dos rebases, o que podemos concluir é que as bases de alguns cenários de integração podem sim ser encontradas, como por exemplo na tabela, cujas informações são que de 476 rebases, foi possível encontrar as bases de 54 cenários (11,34%), houve erro de referência em 275 (ou 57,78%), foi atingido o limite de profundidade 10 vezes (2,1%) e 137 (28,78%) são fast-forward.

Nos testes, existem casos que não são possíveis de ser encontradas bases, seja por causa de uma referência faltando, ou simplesmente porque chegamos ao limite da profundidade de busca do script que foi feito.

Desenvolvendo mais a pesquisa, poderíamos por exemplo, ampliar a amostra com o site de coleta de logs [5], coisa que nos permitiria fazer outras análises neles.

Como exemplo, poderíamos citar a análise do tempo que um desenvolvedor levou para resolver o rebase, a profundidade média dos commits e até os conflitos dos rebases que conseguimos recuperar.

AGRADECIMENTOS

Primeiramente, não poderia deixar de agradecer a meus pais, Marlete Lopes e Cícero Genaldo, cujo suporte me ajudou a me estabelecer meus próprios alicerces. À minha família, que me apoia incondicionalmente.

À Universidade Federal do Cariri e seu corpo docente, que me proporcionaram experiências e conhecimentos novos. Agradeço em especial à Paola Rodrigues de Godoy Accioly, pelo auxílio, pela confiança depositada em mim e pela paciência.

À todos os meus amigos, mas em especial a Gardênia Estevam e Gabriela Queiroga, que me ajudaram a construir e estabelecer minhas bases, me apoiando quando necessário.

Ao CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) pela bolsa e auxílio financeiro, que possibilitou a dedicação ao programa de Iniciação Científica.

A todos que influenciaram, direta ou indiretamente, eu ser quem sou hoje.

REFERÊNCIAS

Accioly, P., Borba, P. & Cavalcanti, G. **Understanding semi-structured merge conflict characteristics in open-source Java projects**. *Empir Software Eng* **23,** 2051–2085 (2018). Acesso em: https://doi.org/10.1007/s10664-017-9586-1. Acesso em: 06/04/2021

[2] CUNHA, Marcela Bandeira. **Entendendo o Uso do Git em Equipes de Desenvolvimento de Software.** 2018. 29f. Tese (Graduação) – Curso de Engenharia da Computação da Universidade Federal de Pernambuco (UFPE), Recife, 2018. Disponível em: https://github.com/predohenr/TG/blob/master/TG-MARCELA_CUNHA.pdf. Acesso em: 16/05/2020

Kalliamvakou, Eirini et al. **The Promises and Perils of Mining GitHub.** University of Victoria, Canadá. p.3-10, 2014. Disponível em: https://kblincoe.github.io/publications/2014_MSR_Promises_Perils.pdf. Acesso em: 06/04/2021

[3] Resultado da execução do script em um repositório genérico. Disponível em: https://imgur.com/a/QgSKo89. Acesso em: 19/05/2020.

[4] Resultado da execução do novo script. Disponível em: https://imgur.com/a/MLgAqbO. Acesso em: 01/07/2020.

[5] Site de Coleta de Logs. Disponível em: https://private-life-of-merge-conflict.firebaseapp.com/ Acesso em: 25/06/2020

[1] Sobre o Git Rebase. Disponível em: https://help.github.com/pt/github/using-git/about-git-rebase. Acesso em: 26/05/2020.