Laboratórios de Informática I
2020/2021

Mestrado Integrado em Engenharia Informática

Sessão Laboratorial 3
Sistemas de Controlo de Versões

O desenvolvimento de *software* é cada vez mais complexo, e obriga a que uma equipa de programadores possa desenvolver uma mesma aplicação ao mesmo tempo, sem se preocuparem com os detalhes do que outros membros dessa mesma equipa estejam a fazer. É evidente que alterações concorrentes (realizadas por diferentes pessoas ao mesmo tempo) podem provocar conflitos quando várias pessoas editam o mesmo bocado de código.

Além disso, não nos devemos esquecer que algumas alterações a um programa, no sentido de corrigir ou introduzir alguma funcionalidade, podem elas mesmas conter erros, e pode por isso ser necessário repor uma versão prévia da aplicação, anterior a essa alteração.

Para colmatar estes problemas são usados sistemas de controlo de versões.

1 Panorama nos Sistemas de Controlo de Versões

Existe um grande conjunto de sistemas que permitem o desenvolvimento cooperativo de *software*. Todos eles apresentam diferentes funcionalidades mas os seus principais objetivos são exatamente os mesmos.

Habitualmente divide-se este conjunto em dois, um conjunto de sistemas denominados de *centralizados*, e um outro de sistemas *distribuídos*:

- Sistemas de controlo de versões centralizados:
  - Concurrent Versions System (CVS): `http://www.nongnu.org/cvs/`;
  - Subversion (SVN): `https://subversion.apache.org/`;
- Sistemas de controlo de versões distribuídos:
  - Git: `http://git-scm.com/`;
  - Mercurial (hg): `http://mercurial.selenic.com/`;
  - Bazaar (bzr): `http://bazaar.canonical.com/en/`;

Estes são apenas alguns exemplos dos mais usados. A grande diferença entre os centralizados e os distribuídos é que, nos centralizados existe um repositório, denominado de servidor, que armazena, a todo o momento, a versão mais recente do código fonte. Por sua vez, nos distribuídos, cada utilizador tem a sua própria cópia do repositório, que podem divergir, havendo posteriormente métodos para juntar repositórios distintos.

2 Instalação do Git

Na disciplina de Laboratórios de Informática I será utilizado o sistema *Git*. Para o instalar siga as instruções em `https://git-scm.com/downloads`.

3 Repositório e Utilizadores

O repositório que será usado na disciplina de LI I está hospedado num serviço designado github (`github.com`).

Cada grupo terá um repositório privado dentro de uma organização criada para esta edição de LI I.

Cada elemento do grupo terá de criar uma conta no github e registar o use username em `https://drive.google.com/file/d/1-TT1o_VsB8rCUSy-hbMSVaQTCpvZIHTv/view?usp=sharing`. Cada aluno deve registar na coluna A o seu username do github. A formação do grupo deve acontecer preenchendo as colunas F e G com o respetivo username github de cada elemento do grupo.

4 Uso do Git

4.1 Clone

A primeira etapa no uso do *git* corresponde ao processo de ir buscar ao servidor a versão mais recente do repositório. **Este processo só deverá ser efetuado uma vez**.

O comando para tal é `git clone` seguido do endereço do repositório. Por exemplo, o utilizador *a1* do grupo deverá executar o seguinte comando.

```
$ git clone https://a1@github.com/umli12021/group1.git
```

Depois de inserir a respectiva *password* correctamente.

O resultado da execução deste comando é uma pasta, criada na pasta atual, com o nome do repositório (`group1` no exemplo acima). Esta pasta será a raiz do repositório. O repositório de cada grupo foi previamente preenchido com pastas e ficheiros, sendo que após o *clone* contém a seguinte estrutura:

```
.
|---- README.md
|---- docs/
      |---- README.md
|---- libs/
      |---- README.md
|---- src/
      |---- README.md
      |---- Main.hs
      |---- Tarefa1.hs
      |---- Tarefa2.hs
      |---- Tarefa3.hs
```

Esta estrutura de ficheiros do repositório é **obrigatória**. Na pasta `src` deve estar o código fonte Haskell, devendo as três tarefas da primeira fase ser implementadas nos três respectivos ficheiros lá colocados.

A pasta `libs` deverá conter bibliotecas que entretanto venha a usar.

A pasta `docs` terá o relatório em LaTeX, a entregar na segunda fase.

O ficheiro `README.md` na raiz contém informação sobre os membros do grupo.

4.2 Status

Um comando extremamente simples, mas bastante útil, designado por *status*, permite ver o estado atual do repositório local (sem realizar qualquer ligação ao servidor).

Depois de editar o ficheiro `README.md` para inserir os dados sobre os elementos do grupo, ao executar o comando `git status` obtemos:

```
    $ git status
Alterações não preparadas para submeter:
  (use "git add <ficheiro>..." para atualizar o que será submetido)
  (use "git restore <file>..." to discard changes in working directory)
        modificado:    README.md
```

Isto indica que neste momento o repositório local tem um ficheiro marcado como editado/modificado.

Se criar um ficheiro de teste, denominado `exemplo.txt`, mas não o adicionar, ao executar o comando *status* obtém-se (além do que já tínhamos):

```
   $ git status
   ...
Ficheiros não controlados:
  (use "git add <ficheiro>..." para incluir no que será submetido)
        exemplo.txt
```

Isto indica que o git não sabe nada sobre aquele ficheiro, e que portanto o irá ignorar em qualquer comando executado.

Para que este ficheiro seja adicionado ao repositório terá que usar o comando `git add exemplo.txt`, tal como usado anteriormente.

Uma boa prática de desenvolvimento é adquirir o hábito de gerir todo o código que programar para o trabalho prático através do sistema de versões.

4.3 Add

Sempre que realizar alterações a um ficheiro e as quiser registar, terá de dar essa informação ao git. Tal como descrito anteriormente, o mesmo comando também é usado para adicionar novos ficheiros ao repositório.

Assim, depois de terminar as alterações a um ficheiro (novo ou não), deve executar o comando git add. Por exemplo, depois de executar a alteração ao ficheiro `README.md` com o seu número de aluno deve executar o comando (sem resultado na consola)

```
    $ git add README.md
```

Se voltar a executar o comando status deverá obeter

```
Alterações para serem submetidas:
  (use "git restore --staged <file>..." to unstage)
        modificado:    README.md
        novo ficheiro: exemplo.txt
```

indicando que ambos os ficheiros estão agora sob controlo do git.

Para registar as alterações e os novos ficheiros ou pastas para no repositório local é necessário realizar um processo designado por *commit* (após o add). Isto poderá ser feito através do comando `git commit`.

```
  $ git commit -m "Modificado o README e criados novos ficheiros de teste"
 2 files changed, 1 insertion(+)
 create mode 100644 exemplo.txt
```

No comando *commit* executado foi adicionada uma opção (`-m`) que é usada para incluir uma mensagem explicativa das alterações que foram introduzidas ao repositório. É boa prática adicionar uma mensagem clara em cada *commit*.

Deve realizar um commit sempre que faça alterações ao seu código que em conjunto formem uma modificação coerente. Os seguintes exemplos podem originar novos commits:

- adicionar uma nova função;
- adicionar um nova funcionalidade;
- corrigir um bug;

Uma boa forma de pensar quando fazer um commit é pensar sobre a mensagem. A mensagem de commit deve ser curta e dizer apenas um coisa. Se a mensagem de commit precisar de uma conjunção, então provavelmente deve fazer dois commits.

Note que depois do commit o código é está no repositório, mas apenas na sua cópia local, ou seja, o seu colega ainda não tem acesso a ele. Veja o comando seguinte para perceber como o colega pode aceder ao código.

4.4 Push

Para enviar as alterações e os novos ficheiros ou pastas para o servidor central é necessário realizar um processo designado por *push*. Isto poderá ser feito através do comando `git push`.

```
  $ git push
To https://github.com/umli12021/group1.git
   27142d6..cc0d148  master -> master
```

Apenas após a execução deste comando o código estará acessível por todos os utilizadores do repositório.

4.5 Pull

Sendo que o *git* é especialmente útil no desenvolvimento cooperativo, vamos ver o que acontece quando um outro utilizador altera o repositório. Suponhamos então que um outro utilizador alterou o ficheiro `README.md`.

Um utilizador de *git* deve atualizar a sua cópia local do repositório sempre que possível e no mínimo antes de iniciar uma sessão de trabalho, para que quaisquer alterações que tenham sido incluídas por outros programadores sejam propagadas do repositório central para a sua cópia local.

Este processo é feito através do comando `git pull`.

Nem sempre os programadores estão a trabalhar em ficheiros distintos. Supondo que dois utilizadores alteraram o mesmo ficheiro em simultâneo, mas em zonas diferentes do ficheiro (por exemplo, cada um modificou apenas o seu nome no ficheiro `README.md`), e que o primeiro já fez *push* da sua alteração, então o segundo irá obter o seguinte resultado ao atualizar a sua cópia.

No entanto, o *git* foi capaz de lidar com a alteração concorrente sem problemas, e portanto, poderá ser feito o *push* destas últimas alterações. :

Em algumas situações poderá acontecer que dois utilizadores tenham editado a mesma zona do ficheiro, e portanto, que o *git* não tenha conseguido juntar as duas versões. Nesta situação, ao realizar o *pull* terá de gerir o conflito manualmente.

Ao editar um ficheiro com conflito aparecerão marcas deste género:

```
codigo haskell muito bom
<<<<<<< HEAD
mais código
=======
mais código do outro utilizador
>>>>>>> branch-a
```

Isto indica a zona com conflito. A parte superior entre as marcas `<<<<<<< HEAD` e `=======` corresponde à versão que está no repositório central. A parte inferior entre as marcas `=======` e `>>>>>>> branch-a` corresponde ao texto produzido pelo utilizador atual (texto na versão local do repositório). Nesta situação é nosso dever remover as marcas (as linhas com `<<<<<<< HEAD` , `=======` e `>>>>>>> branch-a`) e optar por uma das partes (ou então, criar uma nova que resolva o conflito).

Depois de resolver um conflito o utilizador deverá indicar que o conflito foi resolvido:

```
$ git add README.md
$ git commit -m "conflito resolvido"
$ git push
```

4.6 Remove

Vamos agora ver como remover ficheiros do repositório. Isto poderá ser feito através do comando `git remove`, seguido do nome do ficheiro. Para remover o ficheiro `exemplo.txt` faz-se:

```
$ git rm exemplo.txt
   rm 'exemplo.txt'
```

Tal como na operação *add*, temos que fazer *commit* e *push* para que um ficheiro marcado para ser apagado seja efetivamente apagado no repositório central.

Note que embora o ficheiro tenha sido removido da directoria de trabalho, ele ficará guardado no servidor. Portanto se tal for necessário é possível reaver o ficheiro. Se tal for necessário, procure informação sobre o comando `git checkout` em [1].

4.7 Outros comandos úteis

Segue uma lista de alguns comandos

- `git help COMMAND`
- `git log`

Note que para que o `git log` apresente a informação actualizada deverá sempre fazer `git pull` antes.

Referências

Sugere-se a consulta de documentação do *git*, nomeadamente:

[1] `https://rogerdudler.github.io/git-guide/index.pt_BR.html`;

[2] Os comandos disponíveis no *git*, usando o comando `git help`, ou a documentação específica de um comando, com `git help` *comando* em que *comando* é substituído por um dos comandos do *git*.