






Backup Eficiente

Sistemas Operativos

		
Ana Isabel Castro a55522	Joana Miguel a57127	Lúcia Abreu a71634

Grupo 23

Braga, 20 de Maio de 2016

Data de conclusão do projeto	20 de Maio de 2016
Unidade Curricular	Sistemas Operativos
Coordenador da Unidade Curricular	Francisco Coelho Soares Moura
Equipa Docente	Francisco Coelho Soares Moura
	Carlos Miguel Ferraz Baquero Moreno
	José Orlando Roque Nascimento Pereira
	Rui Carlos Mendes Oliveira
	Vítor Francisco Mendes Freitas Gomes Fonte
Ano Letivo	2015/2016

1. Conteúdo

2. Introdução	3
3. Arquitetura da Aplicação	4
4. Backup	7
5. Restore	9
6. Delete	11
7. GC	12
8. Makefile	14
9. Melhorias / Considerações Finais	15

2. Introdução

Backup Eficiente é um projeto realizado no âmbito da unidade curricular de Sistemas Operativos do 2ºano do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

O objetivo deste projeto é construir um sistema eficiente de cópias de segurança – backup – para salvaguardar os ficheiros de um utilizador.

Existem duas operações principais: **backup** e **restore**. O backup cria a cópia dos ficheiros/diretorias indicados e o restore recupera ficheiros/diretorias, repondo para o local original.

Neste projeto, são tidos em conta a **eficiência** e **privacidade**, minimizando o espaço em disco ocupado pelo backup, e criando uma arquitetura cliente/servidor, impedindo o acesso direto do utilizador à diretoria de backup.

O sistema é composto por um programa **cliente** (dá comandos ao sistema) e um programa **servidor** (efetua as operações de cópia e reposição). Possui as funcionalidades de **delete** e **garbage collection**.

3. Arquitetura da Aplicação

A aplicação consiste em construir um sistema eficiente de cópias de segurança para salvar arquivos de utilizador. Para tal construiu-se um modelo **cliente – servidor** em que o cliente envia pedidos de operações ao servidor (serviço de backup). O sistema permite que o cliente peça ao servidor as seguintes operações:

- backup
- restore
- delete
- gc (garbage collection)
- quit

Ao cliente só é permitido requerer uma das operações disponibilizadas pelo servidor sem nunca aceder diretamente às pastas do servidor. O servidor espera que um cliente lhe requeira operações e quando as recebe, gere a forma como as executa, de forma ser eficiente e mantendo a privacidade, não permitindo o acesso direto do utilizador à pasta do dispositivo backup.

No final, informa o cliente se a operação pedida foi bem-sucedida ou não.

Para representar as operações disponibilizadas definiu-se um tipo de dados denominado Operação:

```
typedef enum {  
    BACKUP  
    RESTORE,  
    DELETE,  
    GC,  
    ERRO,  
    QUIT  
} Operacao;
```

O valor ERRO serve para quando o Utilizador pede uma operação inválida. Já o QUIT é uma operação que permite enviar ao servidor para este terminar.

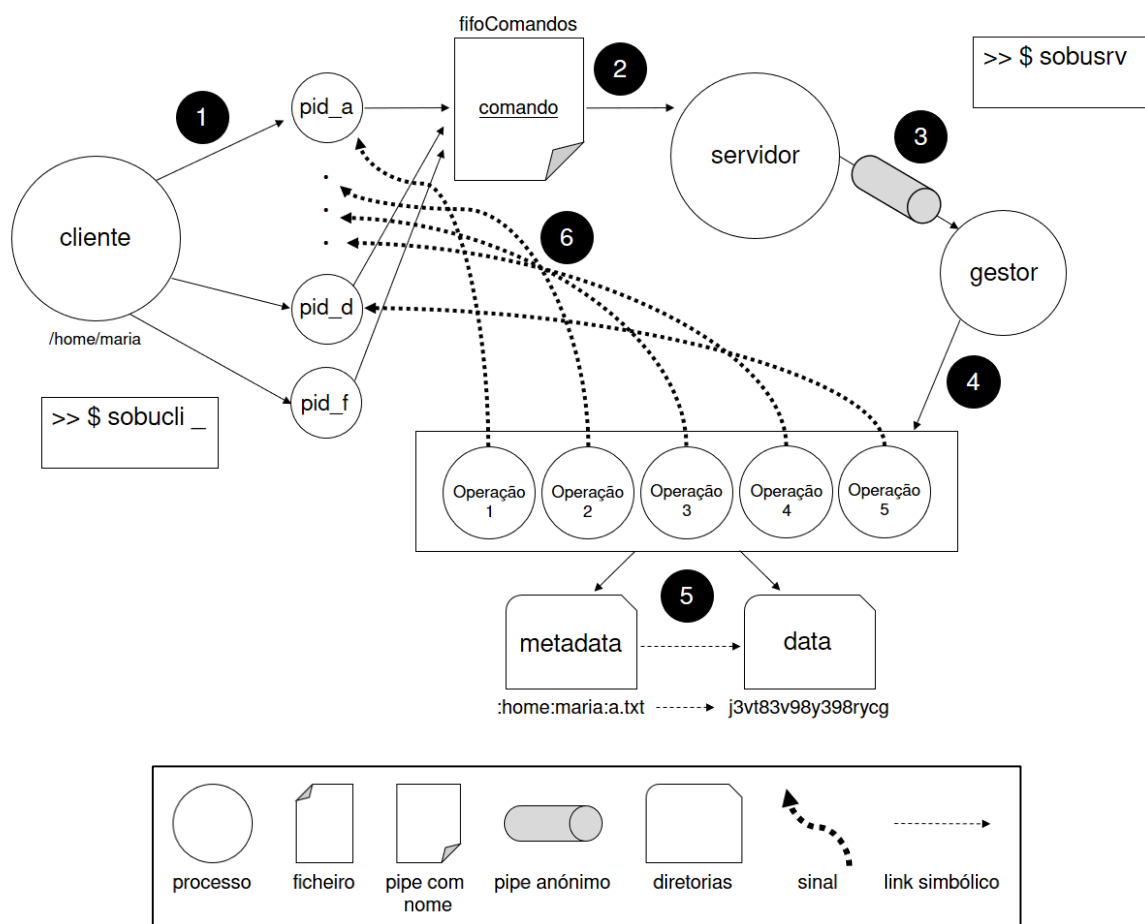
Definiu-se que a estrutura Comando representa um comando único enviado ao servidor. Este pode ou não ter um ficheiro válido associado, dependendo do tipo de operação. Nesse caso, não se preenche os campos ficheiroCaminhoAbsoluto e caminhoAbsoluto.

```
typedef struct sComando {  
    pid_t cliente;  
    Operacao op;  
    char ficheiroCaminhoAbsoluto [PATH_MAX + NAME_MAX];  
    char caminhoAbsoluto [PATH_MAX];  
} Comando;
```

Um **comando** é constituído por:

- **cliente**: pid correspondente ao processo do cliente criado para enviar o comando
- **op**: a operação
- **ficheiroCaminhoAbsoluto**: o caminho absoluto do ficheiro ao qual se pretende realizar a operação (caso seja necessário)
- **caminhoAbsoluto**: o caminho absoluto onde se encontra o ficheiro

O Sistema



1. O cliente faz um **pedido** especificando a **operação** e, eventualmente, um ou mais ficheiros. Para cada um dos ficheiros, ao qual se quer aplicar uma operação, cria-se um novo processo que irá enviar o **comando** ao servidor. Cada um destes novos processos fica à espera de um sinal enviado pelo servidor a informar se o comando solicitado foi bem-sucedido ou não.

Por exemplo:

```
$ sobucli backup a.txt b.txt
```

Este pedido é tratado pelo cliente que cria 2 processos, cada um deles com a função de enviar um comando. Os dois processos ficam a aguardar, cada um deles, por um sinal do servidor com a informação de como correu a operação do ficheiro que têm associado.

2. Para enviar-se o comando ao servidor criou-se um **pipe com nome** chamado “fifoComandos”. Os processos criados pelo cliente escrevem para esse pipe um comando que irá ser lido pelo servidor.

3. O servidor lê do pipe comando a comando e envia-os através de um **pipe anónimo** ao novo processo gestor. A funcionalidade do servidor é de ler os comandos que foram escritos para o pipe “fifoComandos” e de seguida escrever para o pipe anónimo, que serve como canal de comunicação entre o servidor e o seu processo descendente, denominado **gestor**. O servidor bloqueia à espera que o cliente abra o pipe para escrever comandos.

O gestor recebe comandos e cria novos processos para executar cada um dos comandos conforme a operação. Importante referir que este também tem a funcionalidade de gerir o número de comandos a executar simultaneamente, limitando-os a um **máximo de 5**, de modo a não sobrecarregar o sistema. Para tal, utiliza-se um contador que conta o número atual de comandos em execução, e, se for maior que 5, faz-se wait e espera que outro processo termine.

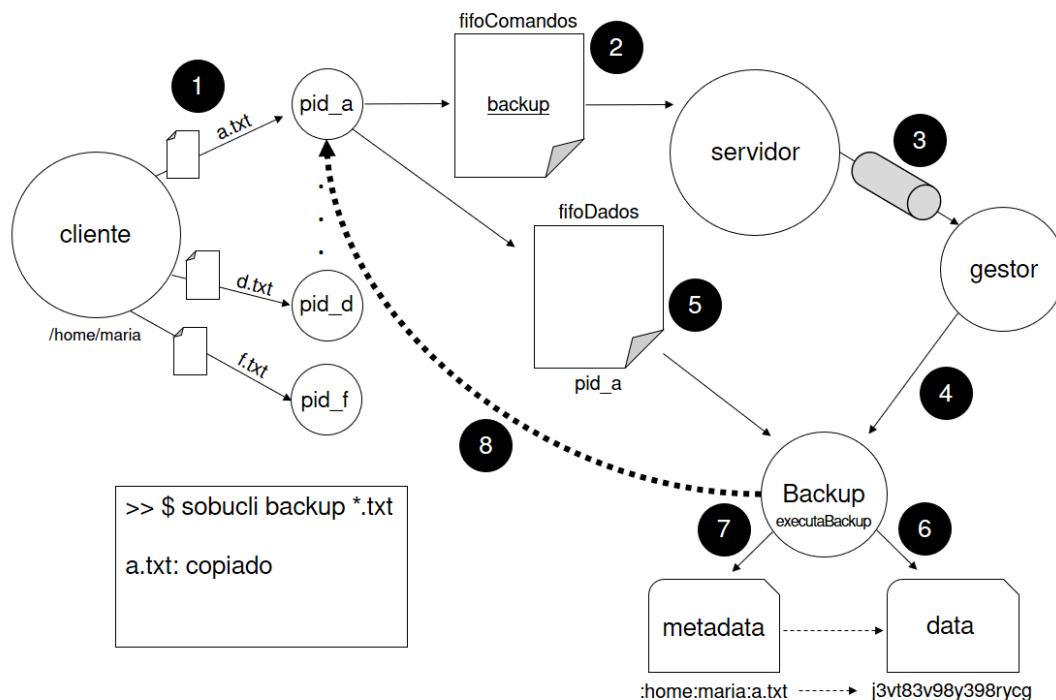
Nota: Quando o servidor não tem comandos para ler, bloqueia e aguarda.

4. O processo gestor cria filhos (novos processos) para executar os comandos. Cada processo executa um comando e só podem ser executados 5 processos concorrentemente. Esse processo é criado com o objetivo de criar um filho para executar a operação do comando e ficar à espera que o filho seja concluído.

5. Dependendo do tipo da operação do comando, atualiza-se ou não, os ficheiros nas pastas metadata e data do servidor.

6. No final da execução do comando o processo operação envia um sinal ao processo do cliente que enviou o comando (este processo é identificado através do seu pid). O envio deste sinal permite informar o cliente se a operação do comando foi bem-sucedida ou não, para aquele ficheiro. Ou seja, é enviado um sinal por cada comando.

4. Backup



1. O cliente faz um pedido especificando que pretende realizar **backup** e os ficheiros a que pretende fazer a cópia de segurança. Na Figura:

```
$ sobucli backup *.txt
```

Ou seja, o cliente pretende fazer backup de todos os ficheiros de texto que possui na diretoria atual.

Este pedido é tratado pelo processo cliente que cria tantos processos quanto o número de ficheiros que se pretende armazenar. Neste caso 6, nomeadamente, a.txt, b.txt, c.txt, d.txt, e.txt e f.txt. São criados 6 processos, cada um deles com a função de enviar um comando ao servidor através de um pipe com nome, que se denominou “fifoComandos”. Ou seja, os 6 processos escrevem nesse pipe, “fifoComandos”.

Assim, este pedido do cliente cria 6 instâncias e, cada uma delas, envia um comando ao servidor (no total são enviados 6 comandos).

2. O processo pid_a, responsável por enviar o comando para fazer backup do ficheiro a.txt, envia através do pipe “fifoComandos” o comando contendo toda a informação necessária ao backup.

Esta informação, contida na estrutura Comando, contém o nome do ficheiro (com o seu caminho absoluto) e o seu local original e, também, o pid do processo pid_a (informação relevante posteriormente ao servidor).

O processo pid_a, após enviar o comando para o pipe “fifoComandos”, cria um novo processo que cria um novo pipe “fifoDados” para escrita e fica à espera que o servidor se ligue para enviar os dados do ficheiro.

O ficheiro é enviado completamente pelo pipe “fifoDados”, o que permite que o servidor - que pode não ter permissões para abrir o ficheiro - nunca abra o ficheiro original, apenas recebe o ficheiro byte a byte.

No fim de enviar o ficheiro o processo pid_a fica a aguardar um sinal do servidor com a informação sobre como correu a operação backup ao ficheiro a.txt.

Entretanto, o processo principal do cliente fica a aguardar pela finalização dos 6 processos que gerou.

3. O servidor lê do pipe “fifoComandos” o comando para efetuar a operação de backup ao ficheiro a.txt e, envia-o através de um pipe anónimo ao processo gestor (criado pelo processo servidor).

4. O processo gestor verifica a quantidade de processos a executar em paralelo, e se for menor que 5, cria um novo processo responsável por executar a operação de backup.

5. O processo executaBackup efetua todos os passos relativos ao backup do ficheiro a.txt. Primeiro abre o pipe com o nome do "pid_a" para leitura e recebe o ficheiro do cliente. Ao mesmo tempo que lê o ficheiro do pipe, escreve o seu conteúdo num ficheiro temporário localmente na pasta .Backup.

6. O servidor fica com uma cópia do ficheiro original a.txt na pasta data. Com a cópia local, calcula o digest do ficheiro a.txt através do programa sha1sum, o que permite verificar se existe algum ficheiro idêntico na diretoria data do servidor.

Caso não exista, o ficheiro é comprimido através do programa gzip, redirecionando do stdin para o ficheiro cópia, e também do stdout para o ficheiro com o nome digest que se encontra na pasta data. Existindo ou não, o ficheiro cópia a.txt é eliminado da pasta data.

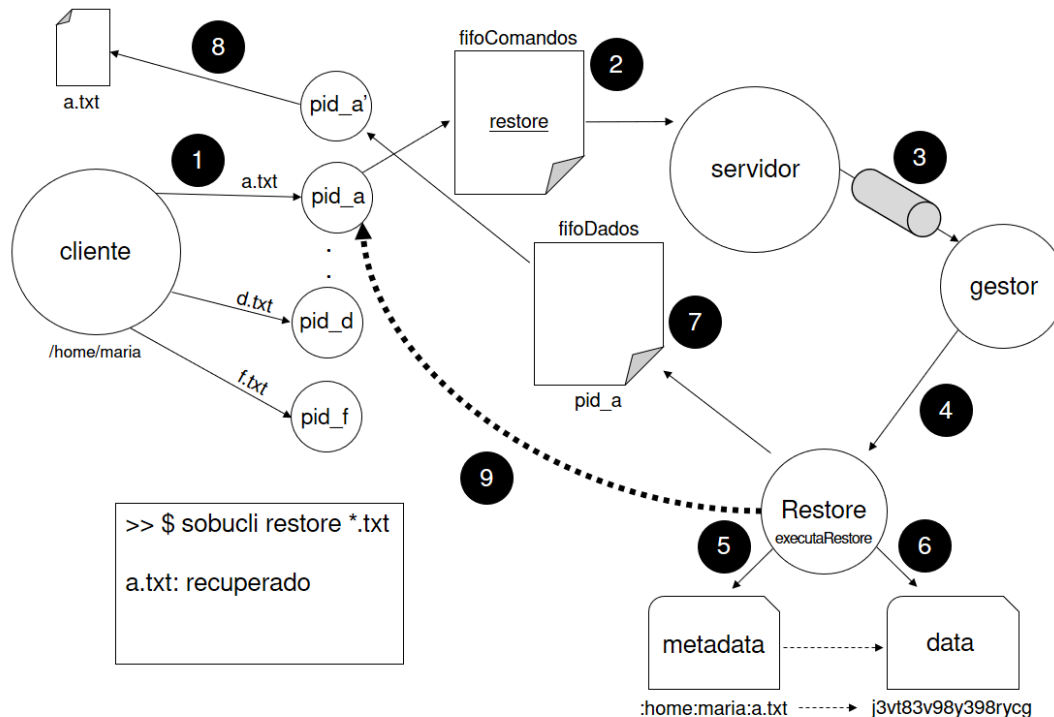
Nota: a compressão de ficheiros permite armazenar os ficheiros de forma mais eficiente.

7. É criado um link simbólico na pasta metadata com o nome do ficheiro absoluto do ficheiro. Ou seja, :home:maria:a.txt, que se liga ao ficheiro da pasta data a que corresponde o seu conteúdo.

Nota: A informação do caminho absoluto do ficheiro é sempre enviada pela estrutura comando. O carácter ‘/’ do caminho é substituído por ‘:’ porque um nome de ficheiro não pode conter esse carácter.

8. No final, o processo responsável por executar a operação de backup do ficheiro a.txt envia um sinal ao processo pid_a que aguarda o sinal. Como correu tudo bem, é enviado um sinal com essa informação e o processo pid_a imprime no ecrã para informar o cliente “a.txt: copiado!”. O processo pid_a termina. Entretanto, o cliente aguarda informação sobre os restantes 5 processos que pediu backup.

5. Restore



1. O cliente faz um pedido (ver figura):

```
$ sobucli restore *.txt
```

Se na diretoria atual em que o cliente se encontra existirem 6 ficheiros txt, irão ser criados 6 processos que irão receber os 6 comandos, respetivos ao restore desses 6 ficheiros.

2. Os comandos irão ser enviados para o servidor, por um pipe com nome, "fifoComandos". Os processos criados pelo cliente irão escrever esses comandos neste pipe e o servidor irá lê-los, bloqueando se não tiver mais comandos para ler.

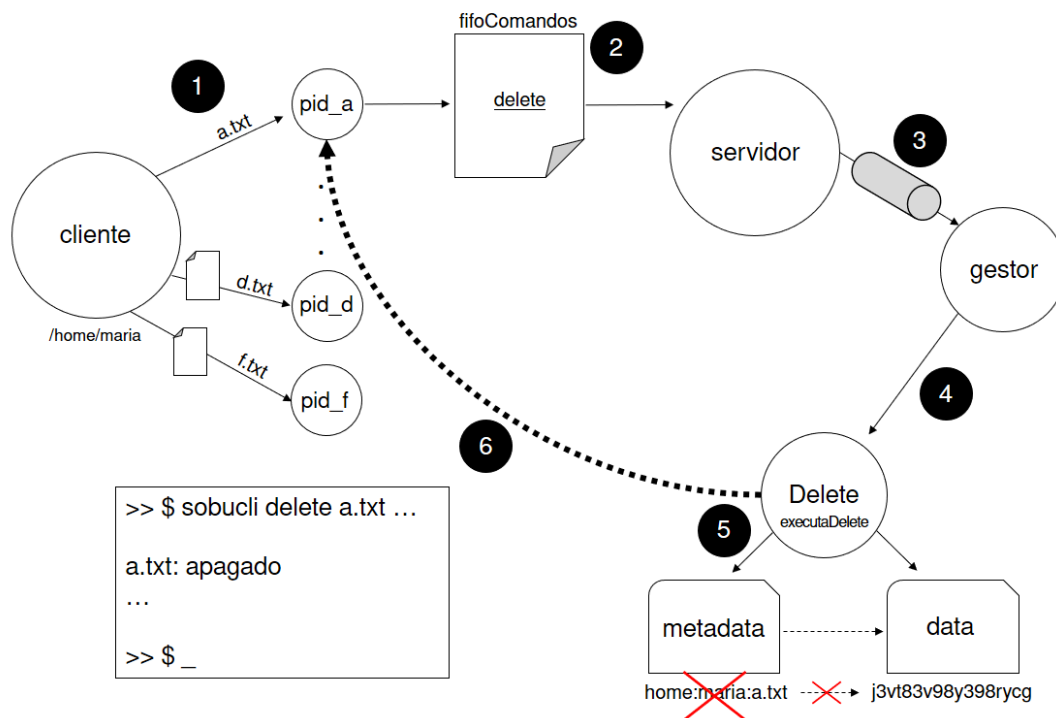
3. O servidor irá criar um novo processo, denominado gestor, e irá enviar-lhe os comandos lidos do pipe "fifoComandos", através de um pipe anónimo.

4. O gestor terá como funcionalidades **enviar** os comandos para serem executados e **gerir** o número de comandos a serem executados. O máximo de comandos que poderão ser executados serão 5, como já foi referido anteriormente.

5. Depois de criado um processo Restore, este primeiro verifica se o ficheiro que queremos descomprimir se encontra na pasta metadata. Se não existe, este irá enviar um sinal ao cliente de que não foi possível restaurar (passo 9). Se existe, teremos que aceder ao ficheiro que se encontra na pasta data para onde o link simbólico (atalho) aponta. Para isso, usou-se a função readlink, que devolve o nome do ficheiro dado o atalho.

6. De seguida, ir-se-á descomprimir o ficheiro com o “gunzip” enviando o seu conteúdo para o pipe com nome “pid_a”, através de um redireccionamento do stdout.
7. O ficheiro será descomprimido e o seu conteúdo irá ser enviado para um pipe com nome criado pelo cliente.
8. O cliente, por sua vez, lê deste pipe e escreve no ficheiro por ele aberto.
9. O processo Restore irá esperar que o seu filho descomprima e envie o ficheiro ao cliente, de modo a informar o Cliente se a operação restore foi bem-sucedida, “a.txt: restaurado”, ou não, “a.txt: não restaurado”.

6. Delete



1. O cliente faz um pedido especificando que pretende fazer a operação delete (ver na Figura):

```
$ sobucli delete a.txt ...
```

Ou seja, o cliente pretende fazer delete na pasta do servidor dos ficheiros passados como argumentos.

Cria processos para cada um dos ficheiros que se pediu para executar a operação. Para eliminar o ficheiro a.txt é criado o processo pid_a.

2. O processo pid_a, responsável por enviar o comando para fazer delete do ficheiro a.txt, envia através do pipe “fifoComandos” o comando contendo toda a informação necessária ao delete.

Entretanto, o processo principal do cliente fica a aguardar pela finalização dos processos que gerou.

3. O servidor lê do pipe “fifoComandos” o comando para efetuar a operação de delete do ficheiro a.txt e, envia-o através de um pipe anónimo ao processo gestor (criado pelo processo servidor).

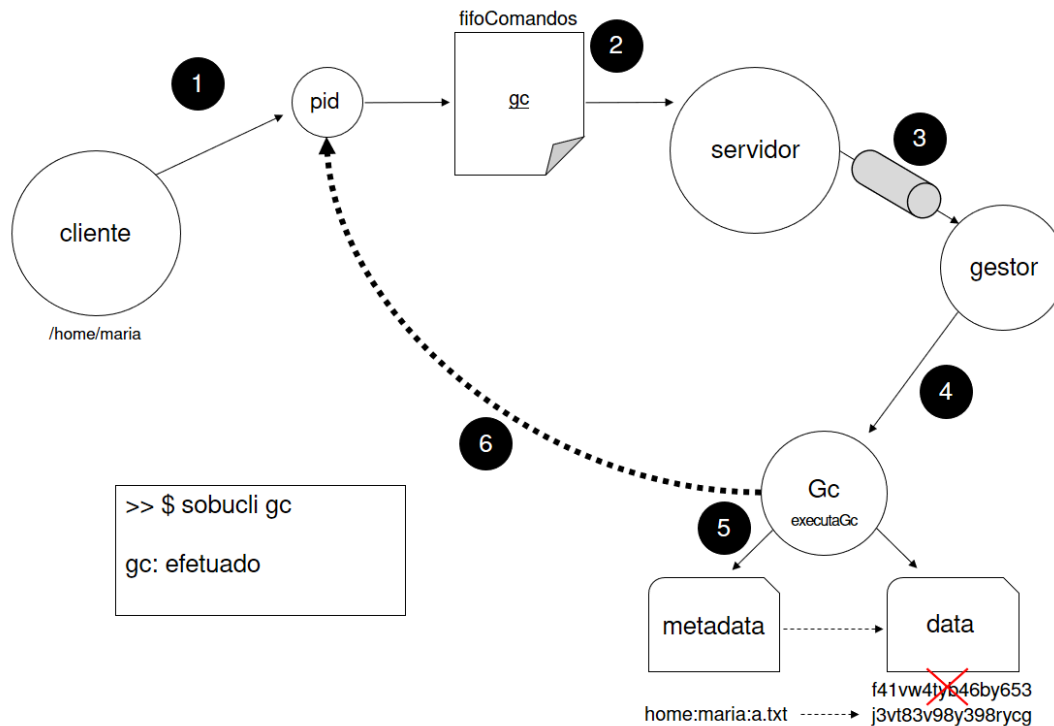
4. O processo gestor verifica a quantidade de processos a executar em paralelo, e se for menor que 5, cria um novo processo responsável por executar a operação de backup ao a.txt. Se não, aguarda.

5. O processo executaDelete efetua todos os passos relativos ao delete do ficheiro a.txt. Na pasta metadata verifica se existe um link simbólico correspondente ao ficheiro a.txt. Se existir, apenas elimina o link simbólico da pasta metadata.

Nota: Apenas é eliminado o link simbólico na pasta metadata (atalho) que aponta para o ficheiro com nome digest na pasta data. Os ficheiros na pasta data são mantidos.

6. Se o ficheiro não existia no servidor, ou se existia e foi eliminado, será enviado um sinal ao processo pid_a dizendo que a operação foi bem sucedida: “a.txt: apagado”.

7. GC



1. O cliente faz um pedido especificando que pretende fazer gc (garbage collection), na figura:

```
$ sobucli gc
```

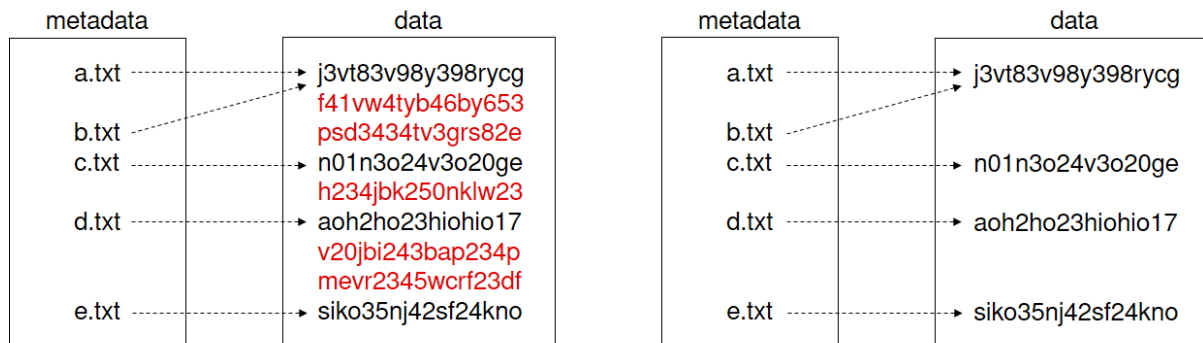
Ou seja, o cliente pretende eliminar todos os ficheiros que se encontram na pasta data do servidor e que não estejam a ser usados por nenhuma entrada em metadata. O cliente cria um processo denominado pid.

2. Este processo irá enviar o comando para o servidor. Para tal, abre-se o pipe com nome "fifoComandos", no qual o processo irá escrever o comando no pipe e o servidor irá lê-lo. Entretanto, o processo principal do cliente fica a aguardar pela finalização dos processos que gerou.

3. O servidor lê do pipe "fifoComandos" o comando para efetuar a operação de gc, e envia-o através de um pipe anónimo ao processo gestor (criado pelo processo servidor).

4. O processo gestor verifica a quantidade de processos a executar em paralelo, e se for menor que 5, cria um novo processo responsável por executar a operação gc.

5. O processo executaGc verifica se para cada ficheiro da pasta data existe um link simbólico para este, na pasta metadata, através do comando “find”. Usou-se a flag “-lname”, para que o find na sua procura, use o nome do ficheiro para onde o link aponta e não o nome do link em concreto.



6. Depois de terminado o executaGc, este envia um sinal ao cliente que fez o pedido, informando-o se o garbage collection foi executado com sucesso ou com insucesso caso alguma chamada ao sistema tenha falhado.

8. Makefile

A makefile está dividida em 5 comandos iniciais, sendo que os restantes comandos executam testes unitários.

make TP: Compila os ficheiros servidor.c e cliente.c gerandos os executáveis sobusrv e sobuccli, respectivamente.

make install: Cria as pasta .Backup na home do utilizador. Dentro desta criou-se a pasta data e metadata. Por fim, move-se os executáveis (sobusrv e sobuccli) para a pasta /usr/local/bin que se encontra na PATH do sistema.

make start: Arranca o servidor em background.

make stop: Termina o serviço de backup.

make clean: Elimina a pasta .Backup da home do utilizadore elimina os executáveis da pasta /usr/local/bin que se encontra na PATH do sistema.

9. Melhorias/Considerações Finais

O sistema Backup eficiente permite ao utilizador realizar as operações de backup, restore, delete e gc (garbage collection). Os requisitos para implementar o sistema foram a **eficiência** e a **privacidade**.

Eficiência

1. O sistema de backup utiliza compressão de dados a fim de minimizar o espaço em disco.
2. Não se guardam ficheiros duplicados, garantindo também a utilização mais eficiente do espaço em disco disponível.
3. Utilizou-se uma arquitetura que permite execução de vários processos concorrentemente
4. A operação gc permite tornar o sistema mais eficiente uma vez que liberta espaço em disco.

Privacidade

1. O cliente nunca acede diretamente à pasta/dispositivo de backup. A comunicação é feita exclusivamente através de pipes com nome.
2. O servidor também respeita a privacidade dos utilizadores uma vez que nunca abre diretamente os ficheiros destes. A transmissão de ficheiros é feita exclusivamente através de pipes com nome. Por exemplo, isto permite que se tenha vários utilizadores a utilizar o cliente com sessão iniciada no sistema, sem que se tenha que dar permissões adicionais ao servidor. Estes utilizadores criam os seus próprios pipes com nome temporários para comunicar com o servidor.

Observações

No sistema não se implementou a funcionalidade que permite fazer backup e restore de diretórias, devido ao tempo disponível. A forma como se abordaria este desafio seria:

1. Diferenciar se estamos a receber ficheiros ou pastas.
2. Se estamos a tentar fazer backup ou restore de uma pasta, é necessário aceder à pasta e colecionar todos os seus ficheiros, recursivamente.
3. Itera-se a coleção e faz-se o backup ou restore, individualmente, que já se encontram implementados.