Transferência rápida e fiável de múltiplos servidores em simultâneo

Trabalho prático Nº2
Comunicações por Computador

Universidade do Minho, Departamento de Informática
Rodrigo Monteiro, Diogo Abreu, e Miguel Gramoso
{a100706, a100646, a100845}@alunos.uminho.pt

1. Introdução

Neste projeto, é implementado um serviço de partilha de ficheiros *peer-to-peer*, em que uma transferência pode ser feita em paralelo por conjuntos de blocos.

Para isso, cada nodo executa uma aplicação designada por `FS_Node`, que é simultaneamente cliente e servidor, e conecta-se a um servidor de registo de conteúdos designado por `FS_Tracker`, informando-o dos seus ficheiros e blocos. Assim, quando um nodo pretende localizar e descarregar um ficheiro, interroga em primeiro lugar o `FS_Tracker`, depois utiliza um algoritmo de seleção de *FS nodes*, e inicia a transferência por UDP com um ou mais nodos, sendo garantida uma entrega fiável.

São utilizadas as seguintes tecnologias: Python3, sqlite3, bind9 e XubunCORE.

2. Arquitetura da solução

2.1. Divisão de ficheiros

Primeiramente, achamos necessário explicitar a nossa abordagem em relação à divisão de ficheiros e gestão de dados.

No `FS_Node`, é utilizada uma classe `File_manager` que é responsável por fazer a divisão dos ficheiros por blocos, de acordo com um determinado *division size*, e por guardar os dados acerca dos ficheiros e dos blocos em estruturas de dados.

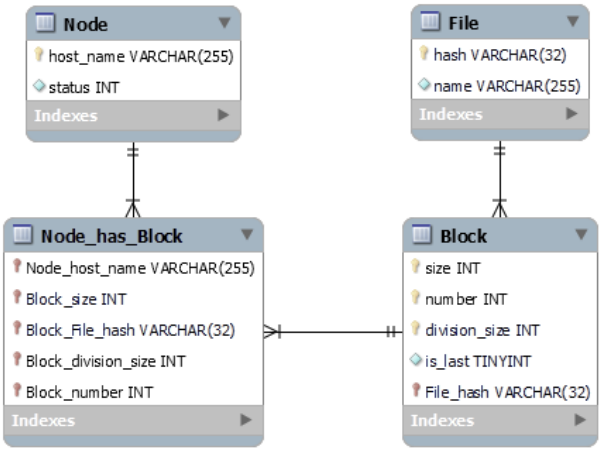
```
Files: { file_name, File ( name, path, hash,
    blocks: { division_size,
      set( Block (division_size, size, number, path, is_last))
    }, is_complete: set(division_size))
}
```

Assim, cada bloco é guardado com as seguintes informações: *division size*, o tamanho que se escolheu para dividir o ficheiro; size, que pode ser igual ao *division size* ou ao resto da divisão; *path* e se é ou não o último bloco.

Cada ficheiro tem um nome, *path*, *hash* única, calculada com *sha1* a partir do conteúdo e do nome do ficheiro, dicionário de *sets* de blocos, em que a *key* é o *division size*, e um *set* de *division sizes* (com uma correspondente divisão completa).

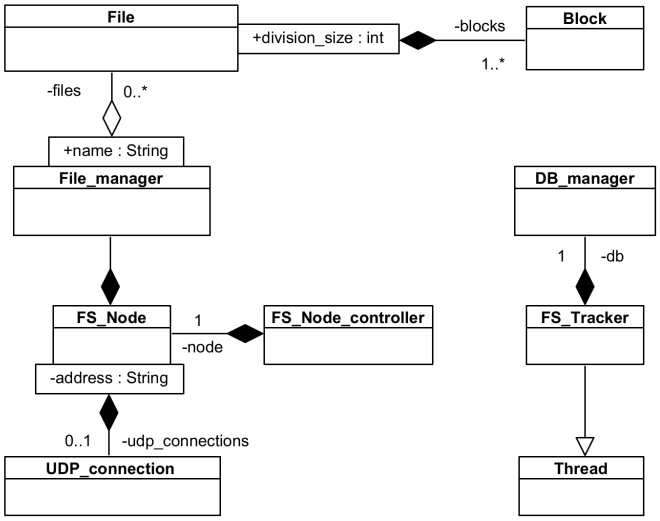
2.2. Base de dados

No `FS_Tracker`, os dados acerca dos ficheiros e dos blocos de cada nodo são guardados numa base de dados `sqlite3`, utilizando a classe `DB_manager`. Escolhemos `sqlite3` pois é de simples utilização e *thread-safe*.



Portanto, os *FS nodes* são identificados a partir do seu *host name*, e os ficheiros são identificados a partir da sua hash única. Para além disso, cada bloco está associado a apenas um ficheiro e presente em N nodos, e cada nodo possui M blocos. Sendo assim, um bloco é identificado com 4 atributos: `size`, `number`, `division_size`, e `File_hash`.

2.3. Diagrama de classes simplificado



Assim, o `FS_Node` também possui um *controller* que permite receber e gerir *input* do utilizador, e um dicionário de `UDP_connection`.

3. FS Tracker Protocol

3.1. Especificação

3.1.1. Atualização parcial e completa dos ficheiros

A seguinte tabela explica a mensagem protocolar utilizada para atualizar o *FS Tracker* acerca dos ficheiros completos de um nodo.

Campo	**Tamanho (bytes)**	**Descrição**
Type	1	Tipo da mensagem enviada (`UPDATE_FULL_FILES = 0`).
Nº of files	2	Número de ficheiros completos.
File hash length	1	Comprimento da hash.
File hash	Variável	Hash do ficheiro em formato binário.
File name length	1	Comprimento do nome do ficheiro.
File name	Variável	Nome do ficheiro em formato binário.
Nº of block sets	1	Quantidade de conjuntos de blocos – um nodo pode ter o mesmo ficheiro dividido de maneiras diferentes, i.e., com tamanhos de divisão diferentes.
Division size	2	Valor do divisor.
Last block size	2	Tamanho do último bloco, ou seja, do resto da divisão, ou o valor do divisor caso o resto seja 0.
Nº of blocks	2	Número total de blocos resultantes da divisão.

A mensagem protocolar utilizada para fazer uma atualização parcial dos ficheiros de um nodo é semelhante à detalhada na tabela acima, sendo que a diferença está no *type* (`UPDATE_PARTIAL = 1`) e no *Nº of blocks* não ser o número de blocos total, mas o número de blocos que serão a seguir identificados no protocolo.

Nº of blocks	2	Número de blocos que serão identificados.
Block number	2	Número do bloco.

Assim, dizer que um bloco é, por exemplo, o nº 3, com uma divisão por 512 bytes, é equivalente a dizer que tem um *offset* de 1536 bytes, e tamanho igual a 512 bytes (ou ao valor do *last block size*).

3.1.2. Resposta genérica

Uma resposta genérica do *FS Tracker*, i.e., que é enviada como resposta associada a mais do que um tipo de pedido (de saída e de atualização). (*type*: `RESPONSE = 8`).

O campo `counter` indica a número da mensagem a que o servidor está a responder (funcionalidade não utilizada na versão final do projeto), e o campo *result status* contém uma das seguintes representações:

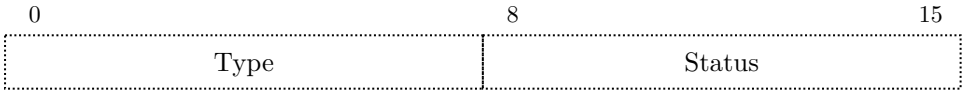
```
class status(IntEnum):
    SUCCESS = 0
    INVALID_ACTION = 1
    NOT_FOUND = 2
    SERVER_ERROR = 3
```

3.1.3. Pedido de saída

Antes de terminar a conexão, o *FS Node* envia um pedido de saída. (`LEAVE = 7`).

3.1.4. Atualização de estado

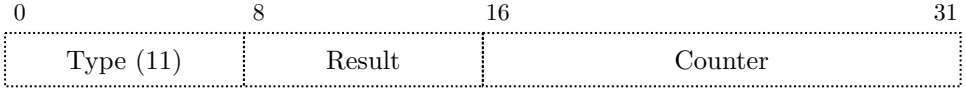
Caso o *FS Node* esteja a enviar blocos para n nodos, então o seu estado equivale ao valor n. (*type:* `UPDATE_STATUS = 3`).



3.1.5. Verificação de estado

Type	1	Tipo da mensagem (`CHECK_STATUS = 4`).
Host name length	1	Tamanho do *host name*.
Host name	Variável	*Host name* em formato binário.

3.1.6. Resposta de estado



3.1.7. Localizar ficheiro por nome

Type	1	Tipo da mensagem (`LOCATE_NAME = 5`).
File name length	1	Tamanho do nome do ficheiro.
File name	Variável	Nome do ficheiro em formato binário.

3.1.8. Localizar ficheiro por hash

Type	1	Tipo da mensagem (`LOCATE_HASH = 6`).
File hash length	1	Tamanho da hash do ficheiro (utilizamos 20 bytes).
File hash	Variável	Hash do ficheiro em formato binário.

3.1.9. Resposta da localização de um ficheiro por nome

Type	1	Tipo da mensagem (`RESPONSE_LOCATE_NAME = 10`).
Nº of host names	2	Número de *host names*.
Host name length	1	Comprimento do *host name* (máximo 255 bytes).
Host name	Variável	*Host name*, em formato binário, de um nodo que possui um ficheiro com o respetivo nome.
Nº file hashes	2	Nº de *file hashes*.
File hash length	1	Comprimento da hash do ficheiro.
File hash	Variável	Hash do ficheiro em formato binário.
Nº host names	2	Número de host names que possuem um ficheiro com o respetivo nome e hash
Host name reference	2	Referência (*index*) relativa aos *host names* listados no início da mensagem.

Exemplo: `(10, 3, 3, "PC1", 3, "PC2", 3, "PC3", 2, 20, <hash1>, 2, 1, 2, 20, <hash2>, 1, 3)` – três nodos possuem um ficheiro com o respetivo nome, no entanto, esse nome está associado a duas `hashes` diferentes, sendo que os nodos `PC1` e `PC2` possuem o ficheiro com a `<hash1>` e o nodo `PC3` possui o ficheiro com a `<hash2>`.

3.1.10. Resposta da localização de um ficheiro por hash

Type	1	Tipo da mensagem (`RESPONSE_LOCATE_HASH = 9`).
Nº of host names	2	Número de *host names*.
Host name length	1	Comprimento do *host name*.
Host name	Variável	*Host name*, em formato binário, de um nodo que possui um ficheiro com a correspondente hash.
Nº of sets	1	Nº de conjuntos de diferentes divisões que o nodo tem para o ficheiro correspondente.
Division size	2	Valor do divisor.
Last block size	2	Tamanho do último bloco.
Full	2	Caso o nodo tenha o ficheiro completo, este campo terá o nº total de blocos, e será o último campo da mensagem. Caso contrário, este campo terá valor 0 e serão adicionados os seguintes campos.
Nº of blocks	2	Número de blocos que serão identificados.
Block number	2	Número do bloco.

Exemplo: `(9, 2, 3, "PC1", 1, 512, 21, 0, 3, 1, 3, 4, 3, "PC2", 1, 1024, 210, 7)` – dois nodos possuem o ficheiro com a hash correspondente, um possui parte do ficheiro, com *division size* igual a 512 bytes, *last block size* igual a 21, e blocos 1, 3, e 4, e o outro possui o ficheiro completo com *division size* igual a 1024 bytes, *last block size* igual 210 bytes, e nº total de blocos igual a 7.

3.2. Implementação

3.2.1. FS_Tracker

O `FS_Tracker` cria uma `thread` por `FS_Node` (`thread-per-client`).

```
while not self.done:
  client, address = self.socket.accept()
  host_name, _, _ = socket.gethostbyaddr(address[0])

  node_thread = Thread(
      target=self.listen_to_client,
      args=(client, host_name)
  )

  node_thread.start()
```

A função `listen_to_client` lê o primeiro byte, e chama uma função *handler* de acordo com o tipo da mensagem recebida. Essa função deserializa a mensagem,

chama uma função que atualiza ou faz uma *query* à base de dados, e envia uma resposta.

Exemplo:

```
def handle_locate_hash_request(self, client, host_name, counter):
        file_hash = self.receive_file_hash(client)

        results, status_db =
          self.db.locate_file_hash(file_hash, host_name)

        if status_db != status.SUCCESS.value:
            self.send_response(client, status_db, counter)
            return

        response = self.encode_locate_hash_response(results, counter)
        client.sendall(response)
```

Neste caso, a função `handle_locate_hash_request` deserializa o resto da mensagem (no código utilizamos os termos `decode` e `encode`), chama uma função da classe `DB_manager` que faz uma *query* à base de dados:

```
def locate_file_hash(self, file_hash, host_name):
  try:
      self.conn.execute("BEGIN")
      query = " ... "
      self.cursor.execute(query, (file_hash, host_name))
      results = self.cursor.fetchall()
      self.conn.commit()
      return results, utils.status.SUCCESS.value
  except Error as e:
      self.conn.rollback()
      return None, utils.status.SERVER_ERROR.value
```

De seguida, caso não tenha ocorrido erro, é chamada a função `encode_locate_hash_response` para serializar a resposta, de acordo com os resultados obtidos e com o protocolo definido. Caso contrário, é enviada uma resposta genérica, com o *status* retornado pela função `locate_file_hash`.

3.2.2. FS_Node

A inicialização do `FS_Node` é feita da seguinte maneira:

```
args = parse_args()
fs_node_1 = FS_Node( ... )
fs_node_1.file_manager.run()

node_controller = FS_Node_controller(fs_node_1)
node_controller_thread = threading.Thread(target=node_controller.run)
node_controller_thread.start()
```

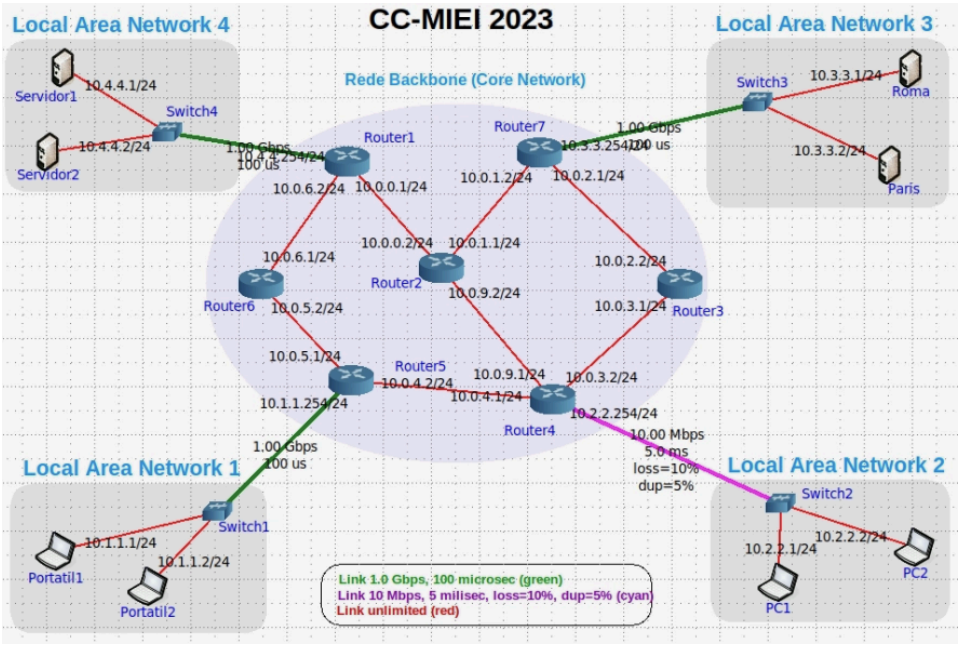
Assim, continuando o exemplo anterior, mas na perspetiva do `FS_Node`, o utilizador decide procurar um ficheiro por hash:

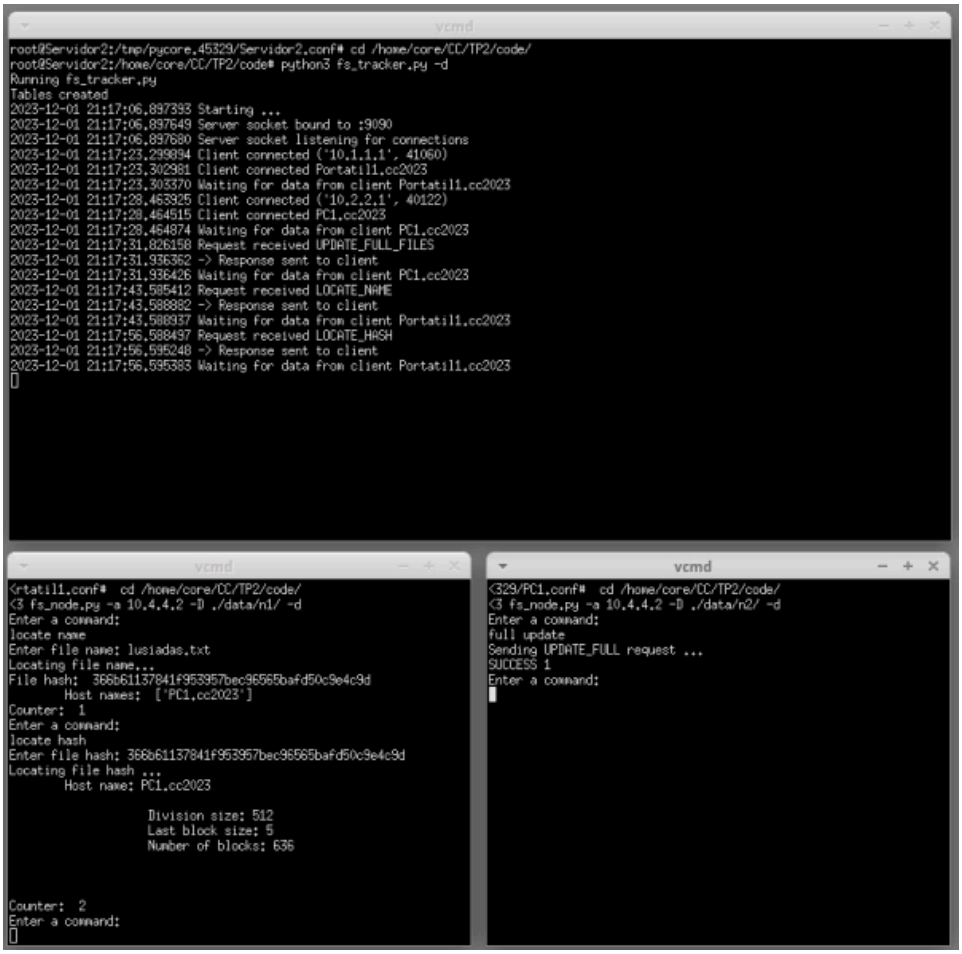
```
elif command == "locate hash" or command == "lh":
    file_hash = input("Enter file hash: ")
    self.node.send_locate_hash_request(file_hash)
    output = self.node.response_queue.get()
    print_locate_hash_output(output)
```

Assim, é chamada a função `send_locate_hash_request`, que chama uma função de serialização, e que envia a mensagem ao `FS_Tracker` para localizar a hash introduzida. De seguida, espera que uma resposta seja adicionada à *queue*. A função que irá adicionar uma resposta à *queue* será a `handle_locate_hash_response` caso não tenha ocorrido um erro no `FS_Tracker`, caso contrário, será a `handle_response`.

O funcionamento é muito semelhante para os outros tipos de `requests`, com exceção dos métodos de serialização e deserialização, que podem ser complexos – o protocolo é eficiente em troca de mais computação na serialização e deserialização.

3.3. Testes





Esta é uma demonstração simples do funcionamento do `FS Track Protocol`, que envolve apenas dois nodos, o "Portatil1" e o "PC1", um servidor de resolução de nomes, "Servidor1", e o *FS tracker*, "Servidor2".

Primeiramente, é iniciado o servidor de resolução de nomes, e de seguida o *FS tracker* com `python3 fs_tracker.py -d` (*debug*). Depois, são iniciados os nodos com `python3 fs_node.py -a 10.4.4.2 -D ./data/n1 -d` e `python3 fs_node.py -a 10.4.4.2 -D ./data/n2 -d`. O nodo "PC1" informa o *FS tracker* de todos os seus ficheiros e blocos com o comando `full update`, e recebe uma resposta com estado `SUCCESS`.

De modo a verificar se o *FS tracker* realmente recebeu e armazenou a informação recebida, o "Portatil1" insere o comando `locate name lusiadas.txt` – um ficheiro que o "PC1" possui. E recebe então uma resposta com a hash e com o *host name* "PC1.cc2023".

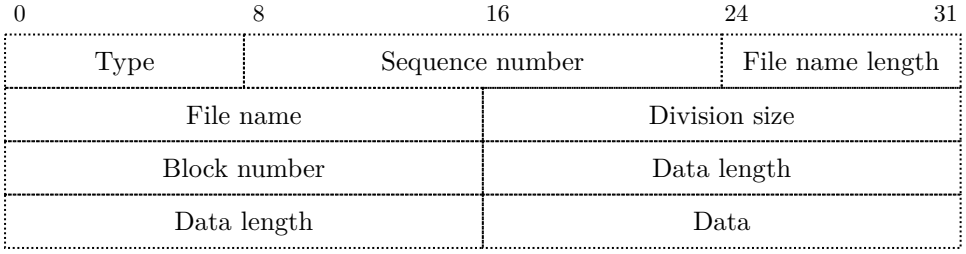
4. FS Transfer Protocol

4.1. Motivação

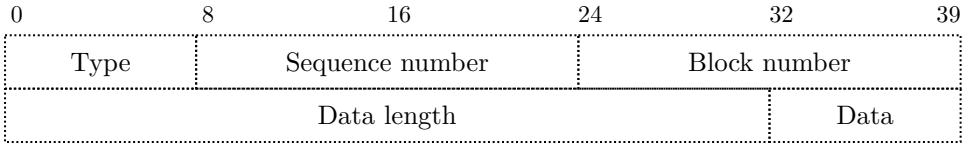
4.2. Vista geral

4.3. Especificação

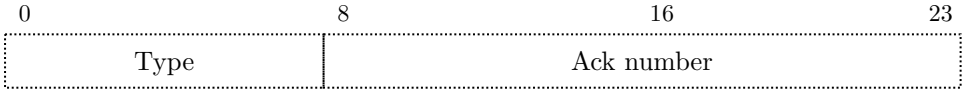
4.3.1. Dados iniciais



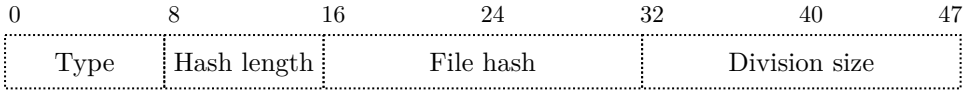
4.3.2. Dados



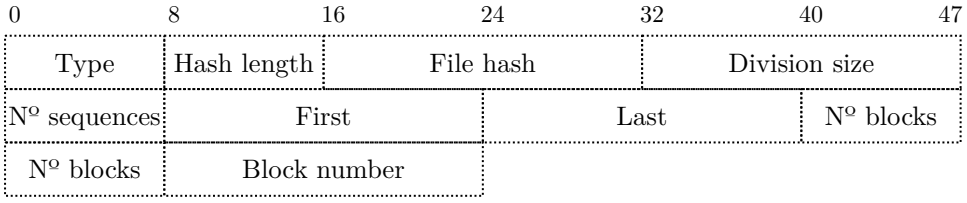
4.3.3. Ack



4.3.4. Pedido de um ficheiro completo



4.3.5. Pedido de parte(s) de um ficheiro



4.4. Implementação

4.5. Testes

5. DNS

6. Conclusões e trabalho futuro

References

1. Norberg, A.: uTorrent transport protocol, https://web.archive.org/web/20161228180615/http://www.bittorrent.org:80/beps/bep_0029.html