

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 se conecta 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.

Para além disso, utiliza 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.

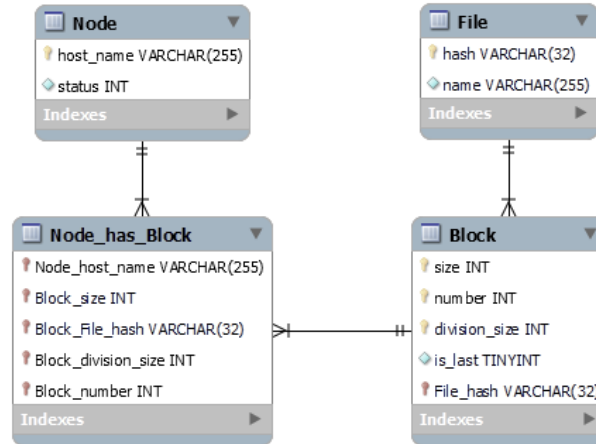
```
Files: { file_name, File ( name, path, hash,  
    blocks: { division_size,  
        set( Block (division_size, size, number, path, is_last))  
    }, is_complete: set(division_size))  
}
```

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).

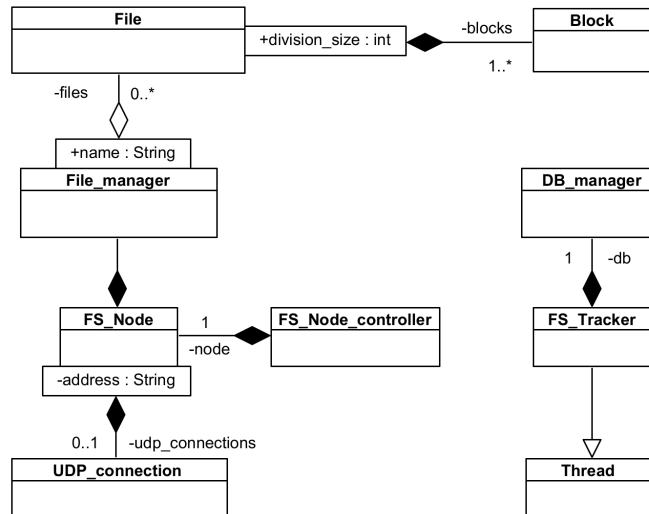
2.2. Base de dados

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).

0	8	24	39
Type	Result status	Counter	

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).

0	7
Type	

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).

0	8	15
Type	Status	

3.1.5. Verificação de estado

Type	1	Tipo da mensagem (CHECK_STATUS = 4).
Host name length	1	Tamanho do <i>host name</i> .
Host name	Variável	<i>Host name</i> em formato binário.

3.1.6. Resposta de estado

0	8	16	31
Type (11)	Result	Counter	

3.1.7. Localizar ficheiro por nome

Type	1	Tipo da mensagem (<code>LOCATE_NAME = 5</code>).
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 (<code>LOCATE_HASH = 6</code>).
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 (<code>RESPONSE_LOCATE_NAME = 10</code>).
Nº of host names	2	Número de <i>host names</i> .
Host name length	1	Comprimento do <i>host name</i> (máximo 255 bytes).
Host name	Variável	<i>Host name</i> , em formato binário, de um nodo que possui um ficheiro com o respetivo nome.
Nº file hashes	2	Nº de <i>file hashes</i> .
File hash length	1	Comprimento da hash do ficheiro.
File hash	Variável	Hash do ficheiro em formato binário.
Nº host names	2	Número de <i>host names</i> que possuem um ficheiro com o respetivo nome e hash
Host name reference	2	Referência (<i>index</i>) relativa aos <i>host names</i> 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 (<code>RESPONSE_LOCATE_HASH = 9</code>).
Nº of host names	2	Número de <i>host names</i> .
Host name length	1	Comprimento do <i>host name</i> .
Host name	Variável	<i>Host name</i> , 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 desserializa 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` desserializa 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 desserialização, que podem ser complexos – o protocolo é eficiente em troca de mais computação na serialização e desserialização.

3.3. Testes

The image displays three terminal windows running the FS_Tracker system. The top window shows the server's startup logs, including client connections and request handling. The bottom-left window shows a client (Portatill) interacting with the server to locate a file hash. The bottom-right window shows another client (PC1) performing a full update and sending a locate hash request.

```
vcmd
root@Servidor2:/tmp/pycore.45329/Servidor2.conf# cd /home/core/CC/TP2/code/
root@Servidor2:/home/core/CC/TP2/code# python3 fs_tracker.py -d
Running fs_tracker.py
Tables created
2023-12-01 21:17:06.897393 Starting ...
2023-12-01 21:17:06.897649 Server socket bound to :3090
2023-12-01 21:17:06.897680 Server socket listening for connections
2023-12-01 21:17:23.298894 Client connected ('10.1.1.1', 41060)
2023-12-01 21:17:23.302981 Client connected Portatill.cc2023
2023-12-01 21:17:23.303370 Waiting for data from client Portatill.cc2023
2023-12-01 21:17:28.463925 Client connected ('10.2.2.1', 40122)
2023-12-01 21:17:28.464515 Client connected PC1.cc2023
2023-12-01 21:17:28.464874 Waiting for data from client PC1.cc2023
2023-12-01 21:17:31.826158 Request received UPDATE_FULL_FILES
2023-12-01 21:17:31.936362 -> Response sent to client
2023-12-01 21:17:31.936426 Waiting for data from client PC1.cc2023
2023-12-01 21:17:43.585412 Request received LOCATE_NAME
2023-12-01 21:17:43.586882 -> Response sent to client
2023-12-01 21:17:43.588937 Waiting for data from client Portatill.cc2023
2023-12-01 21:17:56.588487 Request received LOCATE_HASH
2023-12-01 21:17:56.595248 -> Response sent to client
2023-12-01 21:17:56.595383 Waiting for data from client Portatill.cc2023
[]

vcmd
Portatill.conf# cd /home/core/CC/TP2/code/
C3 fs_node.py -a 10.4.4.2 -D ./data/n1/ -d
Enter a command:
locate name
Enter file name: lusiadas.txt
Locating file name...
File hash: 366b61137841f953957bec96565baf50c9e4c9d
Host names: ['PC1.cc2023']
Counter: 1
Enter a command:
locate hash
Enter file hash: 366b61137841f953957bec96565baf50c9e4c9d
Locating file hash ...
Host name: PC1.cc2023
Division size: 512
Last block size: 5
Number of blocks: 636
Counter: 2
Enter a command:
[]

vcmd
C329/PC1.conf# cd /home/core/CC/TP2/code/
C3 fs_node.py -a 10.4.4.2 -D ./data/n2/ -d
Enter a command:
full update
Sending UPDATE_FULL request ...
SUCCESS 1
Enter a command:
[]
```


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. Especificação

4.1.1. Dados iniciais

Campo	Tamanho (bytes)	Descrição
Type	1	Tipo da mensagem enviada (<code>START_DATA = 0</code>).
Sequence number	2	Número de sequência da mensagem (geralmente 1).
File name length	1	Comprimento do nome do ficheiro.
File name	Variável	Nome do ficheiro em formato binário.
Division size	2	Valor do divisor.
Block number	2	Número do bloco a enviar (pode não ser 1).
Data length	4	Tamanho dos dados a enviar.
Data	Variável	Dados em formato binário

4.1.2. Dados

Type	1	Tipo da mensagem enviada (DATA = 0, ou END_DATA = 1).
Sequence number	2	Número de sequência.
Block number	2	Número do bloco a enviar.
Data length	4	Tamanho dos dados a enviar.
Data	Variável	Dados em formato binário

4.1.3. Ack

0	8	23
Type	Ack number	

4.1.4. Pedido de um ficheiro completo

Type	1	Tipo da mensagem enviada (GET_FULL_FILE = 0).
File hash length	1	Comprimento da hash do ficheiro.
File hash	Variável	Hash do ficheiro em formato binário.
Division size	2	Valor de <i>division size</i> escolhido.

4.1.5. Pedido de parte(s) de um ficheiro

Optamos por incluir sequências neste protocolo para transmitir a mesma quantidade de informação com uma quantidade significativamente menor de bytes.

Type	1	Tipo da mensagem enviada (GET_PARTIAL_FILE = 5).
Hash length	1	Comprimento da hash do ficheiro
File hash	Variável	Hash do ficheiro em formato binário.
Division size	2	Valor de <i>division size</i> escolhido.
Nº of sequences	1	O pedido pode conter [0, N] sequências de blocos.
First	2	Número do primeiro bloco da sequência.
Last	2	Número do último bloco da sequência.
Nº of blocks	2	Número de blocos que não estão presentes nas sequências.
Block number	2	Número do bloco.

4.2. Implementação

Cada nodo possui uma *thread* que fica a “ouvir” por mensagens UDP, uma classe *UDP_receiver_connection* – que guarda dados acerca de uma transferência em curso, gere os números de sequência recebidos e retorna o *ack* que é necessário enviar, – um dicionário de *queues* de *acks* que as *threads* responsáveis por enviar blocos utilizam, e um algoritmo de seleção de nodos.

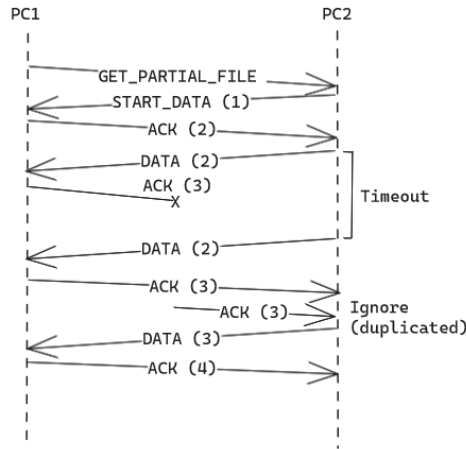
A função que determina o próximo *ack number* verifica se o número de sequência recebido é o esperado, e retorna um *ack number* de acordo com números de sequência que recebeu. Por exemplo, caso esteja à espera do número de sequência 5, e receba os números 7, 8 e 9, irá retornar 6 até o receber, e quando o receber irá retornar 10.

```
def ack(self, seq_num, block_number, is_last, data):
    if (seq_num > self.cur_seq_n and seq_num <= self.cur_seq_n + self.buff):
        self.seq_nums.add(seq_num)
        for n in sorted(self.seq_nums):
            if n == self.cur_seq_n + 1:
                self.cur_seq_n += 1
                self.seq_nums.remove(n)
            else:
                break # out of order seq_num
    return self.cur_seq_n + 1
```

Quando recebe um pedido de transferência completa ou parcial de um ficheiro, é criada uma nova *thread* que corre a função *send_udp_blocks*, que envia um bloco e espera por um *ack* com um limite de 500 ms. Caso o *ack* não seja recebido dentro do limite de tempo, o bloco é enviado novamente. Caso seja recebido, e seja o esperado, é enviado o próximo bloco.

```
self.udp_socket.sendto(packet, address)
expected_ack = seq_num + 1
while max_timeout_retries:
    received_ack = self.udp_ack_queue.get(address, self.udp_ack_timeout)
    if received_ack is not None:
        if received_ack == expected_ack:
            break
    else:
        max_timeout_retries -= 1
        self.udp_socket.sendto(packet, address)
```

Para efetuar a transferência de um ficheiro, o utilizador usa o comando *get* e introduz a hash, depois, o *controller* interroga o *FS Tracker* acerca dos nodos que possuem um ficheiro com essa hash. Caso existam, chama uma função que seleciona nodos e que encarrega blocos a esses nodos – utilizamos um algoritmo simples tenta atribuir a mesma quantidade de blocos a cada nodo. De seguida, são enviados os pedidos de transferência de partes de ficheiros.



4.3. Testes

Três nodos têm o ficheiro `lusiadas.txt`, ou seja, todos menos o `Portatill1`. Assim, de modo a testar a transferência por UDP, executou-se o comando `get` no `Portatill1`, o que originou três transferências em paralelo, cada uma de 213 blocos de 512 bytes. Quando uma das transferências acaba, o *FS Tracker* é atualizado, e, quando todas acabam, os blocos são unidos num só ficheiro.

```

vcmd
>>> Received ack with ack_num 213
>>> [ received 213 == expected 213 ]
leave
Leaving ...
Shutting down ...
def python3 fs_node.py -a 10.4.4.2 -B ./data/hd/ -d
Enter a command:
fu
Sending UPBITE_FULL request ...
SUCCESS 1
Enter a command:
[]

vcmd
2023-12-02 00:53:00.505618 Client connected ('10.2.2.1', 40238)
2023-12-02 00:53:00.506232 Client connected PC1.cc2023
2023-12-02 00:53:00.506670 Waiting for data from client PC1.cc2023
2023-12-02 00:53:00.521395 Request received UPBITE_FULL_FILES
2023-12-02 00:53:00.518412 -> Response sent to client
2023-12-02 00:53:00.518475 Waiting for data from client PC1.cc2023
2023-12-02 00:53:10.832542 Client connected ('10.1.1.1', 32336)
2023-12-02 00:53:10.832941 Client connected Portatill1.cc2023
2023-12-02 00:53:10.833139 Waiting for data from client Portatill1.cc2023
2023-12-02 00:53:25.762794 -> Response sent to client
2023-12-02 00:53:25.762794 -> Response sent to client
2023-12-02 00:53:25.763083 Waiting for data from client Portatill1.cc2023

vcmd
Leaving ...
SUCCESS 2
Shutting down ...
def python3 fs_node.py -a 10.4.4.2 -B ./data/hd/ -d
Enter a command:
fu
Sending UPBITE_FULL request ...
SUCCESS 1
Enter a command:
[]

vcmd
Enter a command:
get
Enter file name: lusiadas.txt
Locating file name...
File hash: 368a61137841f963957bec96565baf50c9e4c3d
List names: ['PC1.cc2023', 'Portatill2.cc2023', 'PC1.cc2023']
Enter a command:
get
Enter file hash: 368a61137841f963957bec96565baf50c9e4c3d

vcmd
>>> [ received 210 == expected 210 ]
>>> Sending packet: (210, 422, False)
>>> [ received 210 == expected 210 ]
>>> [ (timeout) Resending packet: (210, 422, False) ]
>>> Received ack with ack_num 211
>>> [ received 211 == expected 211 ]
>>> Sending packet: (211, 423, False)
>>> [ (timeout) Resending packet: (211, 423, False) ]
>>> Received ack with ack_num 212
>>> [ received 212 == expected 212 ]
>>> Sending packet: (212, 424, True)
>>> Received ack with ack_num 213
>>> [ received 213 == expected 213 ]

vcmd
2023-12-02 00:58:10.721142 Request received LOCITE_MSG
2023-12-02 00:58:10.722843 -> Response sent to client
2023-12-02 00:58:10.724040 Waiting for data from client Portatill1.cc2023
2023-12-02 00:58:11.123213 Request received UPBITE_PARTIAL
2023-12-02 00:58:11.143436 -> Response sent to client
2023-12-02 00:58:11.143483 Waiting for data from client Portatill1.cc2023
2023-12-02 00:58:24.516822 Request received UPBITE_PARTIAL
2023-12-02 00:58:24.518339 -> Response sent to client
2023-12-02 00:58:24.518889 Waiting for data from client Portatill1.cc2023
2023-12-02 00:58:24.520383 Request received UPBITE_PARTIAL
2023-12-02 00:58:24.527152 -> Response sent to client
2023-12-02 00:58:24.534550 Waiting for data from client Portatill1.cc2023

vcmd
>>> Sending packet: (210, 634, False)
>>> Received ack with ack_num 211
>>> [ received 211 == expected 211 ]
>>> Sending packet: (211, 635, False)
>>> Received ack with ack_num 212
>>> [ received 212 == expected 212 ]
>>> Sending packet: (212, 636, True)
>>> Received ack with ack_num 213
>>> [ received 213 == expected 213 ]

vcmd
>>> Sending ack with ack_num 212
>>> Received UDP packet from 10.2.2.2:9190 with flag END_BNf
>>> Received packet: (212, 424)
>>> Sending ack with ack_num 213
>>> Received UDP packet from 10.2.2.1:9190 with flag END_BNf
>>> Received packet: (212, 212)
>>> Sending ack with ack_num 213
File joined successfully
Enter a command:

vcmd
>>> Sending packet: (211, 211, False)
>>> Received ack with ack_num 211
>>> [ (probably duplicated) received 211 < expected 212 ]
>>> [ (timeout) Resending packet: (211, 211, False) ]
>>> Received ack with ack_num 212
>>> [ received 212 == expected 212 ]
>>> Sending packet: (212, 212, True)
>>> Received ack with ack_num 213
>>> [ received 213 == expected 213 ]

```

5. DNS

5.1. Forward DNS bind9 configuration

```
$TTL 86400
@           IN      SOA    Servidor1.cc2023.  admin.cc2023. (
                        1          ; Serial
                        604800     ; Refresh
                        86400      ; Retry
                        2419200    ; Expire
                        86400 )    ; Negative Cache TTL
@           IN      NS     Servidor1.cc2023.
Servidor1   IN      A      10.4.4.1
Servidor2   IN      A      10.4.4.2
Portatil1   IN      A      10.1.1.1
Portatil2   IN      A      10.1.1.2
PC1         IN      A      10.2.2.1
PC2         IN      A      10.2.2.2
Roma        IN      A      10.3.3.1
Paris       IN      A      10.3.3.2
```

5.2. Reverse DNS bind9 configuration

```
$TTL 86400
@           IN      SOA    Servidor1.cc2023.  admin.cc2023. (
                        1          ; Serial
                        604800     ; Refresh
                        86400      ; Retry
                        2419200    ; Expire
                        86400 )    ; Negative Cache TTL
@           IN      NS     Servidor1.cc2023.
1.1.1       IN      PTR    Portatil1.cc2023.
2.1.1       IN      PTR    Portatil2.cc2023.
1.2.2       IN      PTR    PC1.cc2023.
2.2.2       IN      PTR    PC2.cc2023.
1.3.3       IN      PTR    Roma.cc2023.
2.3.3       IN      PTR    Paris.cc2023.
1.4.4       IN      PTR    Servidor1.cc2023.
2.4.4       IN      PTR    Servidor2.cc2023.
```

5.3. Default zones

```
zone "cc2023" {
    type master;
    file "/etc/bind/cc2023";
};

zone "10.in-addr.arpa" {
    type master;
    file "/etc/bind/cc2023reverse";
};
```

6. Conclusões e trabalho futuro

Para concluir, achamos que conseguimos implementar os requisitos fornecidos no enunciado, e por conseguinte, conseguimos entender o funcionamento de *sockets* UDP e TCP, como construir um protocolo eficiente em formato binário, como lidar com perdas e duplicados numa transferência por UDP, como configurar um servidor DNS, entre outros. No entanto, existe um conjunto de funcionalidades que gostaríamos de ter adicionado:

- Utilização de um método `noack` para a transferência de ficheiros;
- Solução para o caso de um nodo deixar de estar disponível a meio de uma transferência;
- Implementação de um `checksum` nos pacotes enviados por UDP – não achamos estritamente necessário implementar um, pois o protocolo UDP já possui `checksum`.

References

1. BIND9 Administrator Reference Manual, <https://bind9.readthedocs.io/en/latest/index.html>
2. Kurose, J., Ross, K.: Computer Networking: A Top-Down Approach. (2020)
3. Norberg, A.: uTorrent transport protocol, http://www.bittorrent.org/beps/bep_0029.html