



UNIVERSIDADE FEDERAL DE SANTA CATARINA - UFSC  
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA  
CURSO DE CIÊNCIAS DA COMPUTAÇÃO  
INE5418 - Computação Distribuída  
Professor Odorico Machado Mendizabal

## Relatório T1: Sistemas P2P

Mariana Mazzo Heitor (20203088)  
Raquel Cristina Schaly Behrens (20100544)

28 de outubro de 2024

# 1. Execução

É necessário ter *python3* instalado na(s) máquina(s).

Para rodar, digite o seguinte comando:

*python3 initialize.py*

A seguir, o console permite ao usuário escolher: quais nós ele vai inicializar naquele computador; o nó que começará a busca e o arquivo.p2p com os metadados necessários para a transferência de um arquivo.

```
Digite uma lista de IDs dos nós que vão estar executando nesse computador, se
parados por vírgula: 0, 1, 2, 3, 4

Digite 'sair' para interromper a execução, ou então digite o nó que vai procu
rar o arquivo e o arquivo .p2p desejado!
Exemplo: se quero começar a busca pelo nó 0, e o arquivo .p2p é o image.png.p
2p, digito: '0 image.png.p2p'
0 image.png.p2p
```

É possível ativar todos os logs de debug no arquivo *node.py*, colocando *PRINT\_LOGS* como *True*!

Caso *PRINT\_LOGS* permaneça como *False*, conforme os arquivos forem sendo transferidos, a taxa de transferência dos chunks e a taxa de transferência total do arquivo aparece no console.

```
Chunk IMAGE.PNG.CH0 100.00% transferred.
PERCENTAGE OF THE FILE ALREADY TRANSFERRED: 25.0%
Chunk IMAGE.PNG.CH1 6.67% transferred.
Chunk IMAGE.PNG.CH1 13.33% transferred.
Chunk IMAGE.PNG.CH1 20.00% transferred.
Chunk IMAGE.PNG.CH1 26.67% transferred.
Chunk IMAGE.PNG.CH1 33.33% transferred.
Chunk IMAGE.PNG.CH1 40.00% transferred.
Chunk IMAGE.PNG.CH1 46.67% transferred.
Chunk IMAGE.PNG.CH1 53.33% transferred.
Chunk IMAGE.PNG.CH1 60.00% transferred.
Chunk IMAGE.PNG.CH1 66.67% transferred.
Chunk IMAGE.PNG.CH1 73.33% transferred.
Chunk IMAGE.PNG.CH1 80.00% transferred.
Chunk IMAGE.PNG.CH1 86.67% transferred.
Chunk IMAGE.PNG.CH1 93.33% transferred.
Chunk IMAGE.PNG.CH1 100.00% transferred.
PERCENTAGE OF THE FILE ALREADY TRANSFERRED: 50.0%
```

Após isso, o usuário pode escolher outro nó que começará a busca e outro arquivo .p2p com os metadados necessários para a transferência de um arquivo.

Para parar a execução do programa, basta digitar “sair”.

## 2. Decisões e Estratégias de Implementação

### 2.1. Criação e Inicialização dos Nodos

Optamos por permitir ao usuário a criação de mais de um nodo em um mesmo terminal. Dessa forma, é possível tanto a execução de todos os nodos com apenas um terminal quanto a execução de um nodo por terminal, permitindo diversas possibilidades de configuração.

É requisitado que o usuário informe quais nodos ele deseja criar naquela instância de programa específica ao inicializar a aplicação, como detalhado na seção ‘Execução’ desse relatório. Com isso, a partir do arquivo ‘topologia.txt’, é criado um objeto *Node* dos nodos solicitados pelo usuário e de sua vizinhança. Em seguida, com o arquivo ‘config.txt’, são configurados os principais atributos - IP, porta UDP e taxa de transferência - para os nodos criados previamente.

### 2.2. Definição do Arquivo e do Nodo Requisitante

Devido à possibilidade de criar mais de um nodo em uma mesma instância de programa, pode ser necessário que o usuário defina qual será o ‘nodo principal’, isto é, o nodo que será responsável por requisitar os chunks para a concatenação do arquivo. Para isso, optamos por solicitar que o usuário informe no console, como explicado na seção ‘Execução’ desse relatório. Caso o *input* do usuário contenha apenas um parâmetro, consideramos que se trata do arquivo metadado que será utilizado para o processo de transferência e utilizamos o primeiro nodo criado - o primeiro id, dentre os escolhidos pelo usuário, a aparecer no arquivo ‘topologia.txt’ - para ser o ‘nodo principal’. Assim, caso a instância contenha apenas um nodo, ele automaticamente será selecionado, sem a necessidade do usuário informar.

Juntamente com a solicitação do nodo requisitante, é pedido ao usuário que informe o caminho do arquivo metadado que será utilizado na execução do programa. Então, as informações deste arquivo serão coletadas e repassadas às suas devidas funções para auxiliar o nodo na comunicação.

Para garantir que possa ser passado mais de um arquivo para mais de um nodo solicitar durante a execução do programa, esses pedidos de *input* ao usuário se mantêm em um loop ‘*while true*’, que executará até o usuário decidir encerrar o programa. Como os servidores UDP são executados a partir de outras threads criadas,

isso não interferirá na execução da comunicação, podendo se responsabilizar inteiramente por ‘ouvir’ o usuário.

## 2.3. Requisição UDP

Abrir uma requisição UDP envolve dois passos: abrir um ‘servidor’ para cada nodo do programa e abrir um ‘cliente’ para o nodo que deseja enviar uma mensagem. A fim de manter todos os ‘servidores’ executando ao mesmo tempo, é criada uma thread responsável por executar a função *create\_udp\_socket()* da classe *Node* para cada nodo gerado. Essa função ficará em um loop ‘*while true*’ até o encerramento do programa, ‘ouvindo’ e ‘processando’ as mensagens recebidas pelos ‘clientes’ dessa conexão, que, por não precisar ficar rodando o tempo inteiro, optamos por não criar novas threads para executá-los, encerrando a conexão logo após enviar a mensagem pretendida ao ‘servidor’. A funcionalidade do ‘cliente’ é definida na função *create\_udp\_client()* da classe *Node*.

Como mencionado anteriormente, o ‘servidor’ da requisição UDP é tratado pela função *create\_udp\_socket()*. Essa função é responsável por criar um socket de protocolo UDP de domínio *AF\_INET* e tipo *SOCK\_DGRAM*. Com isso, ele estará ativo ‘ouvindo’ os protocolos que chegarem no endereço *{ip:porta}* daquele nodo. Essa ação é executada pela função *handle\_udp\_client()* da classe *Node*, que é responsável por decodificar e tratar as mensagens recebidas pelos ‘clientes’. Para facilitar esse tratamento, decidimos dividir as mensagens em três tipos: ‘*searching\_file*’, ‘*found\_file*’ e ‘*send\_file*’. Cada tipo possui um cabeçalho diferente para a mensagem enviada, apesar de todas estarem em formato *json*, e, portanto, uma forma diferente de lidar com cada. Isso será detalhado no tópico abaixo.

Em relação ao ‘cliente’, executado na função *create\_udp\_client()*, é criado um socket semelhante ao do ‘servidor’, porém é definido um *timeout* para ele executar, gerando uma exceção ao não conseguir enviar a mensagem a tempo. Além disso, outra diferença fundamental é que ele é encerrado logo após sua execução. Sua função se trata basicamente de enviar uma mensagem a um servidor informando o tipo de requisição a ser realizada e os parâmetros correspondentes a ela, que será detalhado no tópico abaixo. O ‘cliente’ tentará enviar essa mensagem *max\_attempts* vezes, variável definida na função para definir a quantidade de tentativas, sendo imprimido no console caso falhe.

## 2.4. Formato das Mensagens Enviadas à Requisição UDP

A função `handle_udp_client()` é que é a responsável por receber as mensagens UDP. Nela são tratados três tipos de requisição:

1. 'searching\_file' (node.py: 104-127): mensagens UDPs enviadas para os nós vizinhos, procurando pelo arquivo. As mensagens desse padrão são dicionários, como na imagem abaixo. Elas contêm também o nome do arquivo procurado, o endereço do nó vizinho que enviou a requisição, o endereço do nó que começou a procura do arquivo (para, ao encontrar o arquivo, enviá-lo diretamente para o nó original), e o flooding (que diminui a cada vizinho).

```
message_sent = {
    'type_client': 'searching_file',
    'file_wanted': file_wanted,
    'address': (self.host, self.port),
    'original_address': (original_address[0], original_address[1]),
    'flooding': flooding-1
}
```

Ao receber essa mensagem, caso encontre arquivos naquele nó, avisa-se o nó original. E, caso o flooding não tenha terminado, continua a busca para os nós vizinhos.

2. 'found\_file' (node.py: 150-164): mensagens UDPs enviadas para o nó que iniciou a busca, avisando que encontrou algum chunk. As mensagens desse padrão são dicionários, como na imagem abaixo. Elas contêm também o nome do arquivo procurado, o nome do chunk encontrado, o endereço do nó que encontrou o chunk, bem como sua taxa de transferência.

```
message_sent = {
    'type_client': 'found_file',
    'file_wanted': file_wanted,
    'files_found': matching_files,
    'address': (self.host, self.port),
    'transfer_rate': self.transfer_rate
}
```

Quando o nó que iniciou a busca recebe essa mensagem, registra esse chunk encontrado para decidir futuramente de quem buscará o chunk.

3. 'send\_file' (node.py: 166-187): mensagens UDPs enviadas para o nó que encontrou o chunk, avisando que ele pode iniciar a conexão TCP para enviar

esse chunk. As mensagens desse padrão são dicionários, como na imagem abaixo. Elas contêm também o endereço do nó que iniciou a busca e o nome do chunk encontrado.

```
message_sent = {  
    'type_client': 'send_file',  
    'address': (self.host, self.port),  
    'file': file  
}
```

Quando o nó que encontrou o chunk recebe essa mensagem, ele começa a tentar se conectar com o socket TCP no nó que iniciou a busca.

## 2.5. Busca e Seleção de Chunks

Como mencionado no tópico anterior, ao encontrar um arquivo é enviada uma mensagem ao ‘servidor’ UDP do nodo requisitante informando quais arquivos foram encontrados e o endereço *{ip:porta}* e a taxa de transferência do nodo no qual foi encontrado, armazenando esses dados - em formato de uma lista contendo essas informações - em um dicionário do nodo requisitante de acordo com a qual parte da imagem o chunk pertence.

Entretanto, pode haver a possibilidade de o próprio nodo requisitante já possuir algum dos chunks necessários para formar a imagem completa. Para resolver esse problema, decidimos criar uma função *configure\_known\_chunks()* na classe *Node* que será invocada somente pelo nodo requisitante ao inicializar os nodos. Essa função funciona de forma semelhante à requisição ‘found\_file’ da requisição UDP, adicionando os nodos encontrados no próprio nodo ao dicionário. Porém, ao invés de utilizar sua própria taxa de transferência, o valor é definido como infinito, simbolizando que chega instantaneamente, uma vez que já se encontra no próprio nodo que requisitou. Essa escolha foi feita para que a taxa seja sempre maior do que a de qualquer outro nodo que possa possuir o mesmo arquivo, sendo sempre o nodo escolhido no algoritmo para selecionar de quem será feita a transferência do arquivo. Esse algoritmo será explicado na sequência.

Ao inicializar os nodos, é criada uma thread para o nodo requisitante que executa a função *search\_chunks()* da classe *Node*. Essa thread foi criada com o intuito de permitir que o nodo possa verificar os chunks recebidos sem a necessidade de interromper a busca no ‘servidor’ UDP durante o processo.

A função *search\_chunks()* é responsável por: verificar os arquivos chunks encontrados; decidir o melhor nodo - dentre os que possuem determinado chunk - para requisitar o arquivo, solicitando a invocação da função responsável pela criação do protocolo TCP para a sua transferência; conferir se é possível montar a imagem completa, isto é, se todos os chunks foram encontrados; e, em caso afirmativo, chamar a função responsável pela concatenação deles. Ela funciona de forma a verificar os arquivos identificados, após receber ao menos um arquivo e ter passado um tempo definido de busca pelos nodos no protocolo UDP, até que todos os chunks sejam encontrados, solicitando a concatenação do arquivo, ou que um *timeout* definido seja excedido, informando a falha ao encontrar todos os chunks nessa situação.

Durante a busca pelos chunks encontrados, é realizada a verificação de qual é o melhor nodo para solicitar determinado arquivo. Essa verificação é feita de forma simples, apenas comparando as respectivas taxas de transferência e selecionando o nodo que possui a com maior valor. Após essa seleção, é verificado se o valor dessa taxa em algum dos casos é igual a infinito. Caso seja, indica que o arquivo já está presente no próprio nodo, não fazendo nada além de informar ao usuário pelo console. Caso contrário, é enviada uma solicitação para iniciar a transferência desse arquivo ao nodo requisitante.

## **2.6. Requisição TCP e Transferência de Arquivos**

Após decidir de quais nós o nó requisitante vai receber os chunks, é checado quais chunks já estão no nó requisitante, e o código prossegue para criar o socket TCP do nó requisitante (node.py: 333-341).

Antes de criar o socket TCP, o nó requisitante envia para o nó do chunk uma mensagem UDP com tipo *'send\_file'*, informando que aquele nó pode enviar para o nó requisitante o chunk escolhido (node.py: 201-212).

O nó do chunk escolhido, então, ao receber essa mensagem UDP, tenta até conseguir criar uma conexão TCP e conectar com o nó requisitante para enviar o chunk. Decidimos implementar um ping, porque pode acontecer de a conexão TCP do nó requisitante não estar aberta ainda (node.py: 174-178) .

Para transferir os arquivos com a velocidade total de envio do nó remetente, definiu-se o tamanho máximo da mensagem a ser enviada. Por isso, o chunk é enviado em blocos do tamanho máximo, e foi adicionado um *time.sleep()* que espera

o tempo necessário para enviar cada chunk, calculado por (tamanho máximo da chunk / taxa de transferência) (node.py: 248-286).

O nó do chunk receptor define um tamanho máximo por envio de chunks, que é 1024 bytes, definido no início de *node.py* em *MAX\_REQ\_RECV*, e salva os blocos recebidos em um arquivo com o mesmo nome do arquivo procurado (node.py: 66-80).

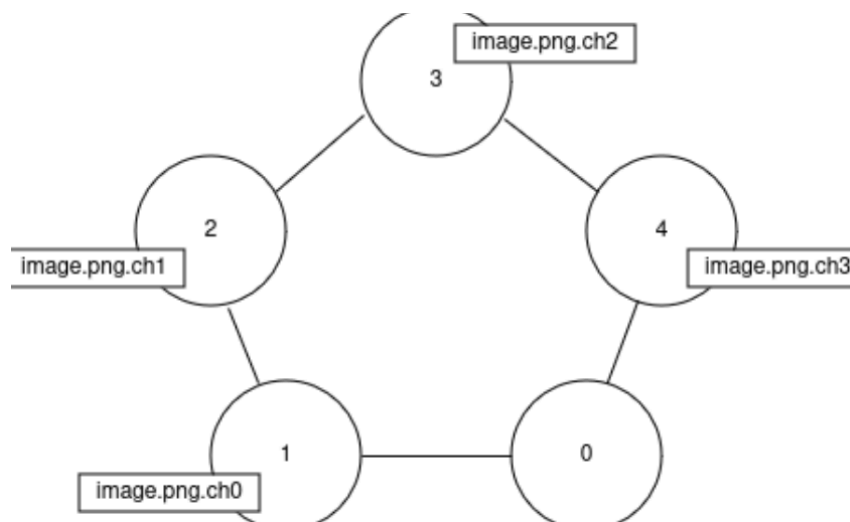
## 2.7. Concatenação de Arquivos

Caso aconteça algum problema na transferência de arquivos, ou estoure o timeout definido no início do arquivo *node.py*, e isso resulte em não existir todos os chunks no nó receptor, mostra-se um erro para o usuário informando que não encontrou todos os chunks (node.py: 350-251).

Se todos os chunks estiverem no nó receptor, os chunks são mesclados em um único arquivo com o nome inicial dos chunks (nome antes da extensão .ch) (node.py: 357-378).

## 3. Exemplos de Saída

Com os logs de debug ligados, e com a seguinte configuração:



Exemplo 1:

- Estado inicial:



```
≡ config.txt X
config > ≡ config.txt
1 0: 127.0.0.1, 6000, 200
2 1: 127.0.0.1, 6001, 200
3 2: 127.0.0.1, 6002, 200
4 3: 127.0.0.1, 6003, 200
5 4: 127.0.0.1, 6004, 200
6
```

```
▼ nodes
  ▼ 0
  ▼ 1
  | ≡ image.png.ch0
  ▼ 2
  | ≡ image.png.ch1
  ▼ 3
  | ≡ image.png.ch2
  ▼ 4
  | ≡ image.png.ch3
```

- Parâmetros:

Nó 0 quer receber image.png

```
$ python3 initialize.py
Digite uma lista de IDs dos nós que vão estar executando nesse computador, separados por vírgula: 0, 1, 2, 3, 4
Node 1 listening UDP in 127.0.0.1:6001.
Node 0 listening UDP in 127.0.0.1:6000.
Node 2 listening UDP in 127.0.0.1:6002.
Node 3 listening UDP in 127.0.0.1:6003.

Digite 'sair' para interromper a execução, ou então digite o nó que vai procurar o arquivo e o arquivo .p2p desejado!
Exemplo: se quero começar a busca pelo nó 0, e o arquivo .p2p é o image.png.p2p, digito: '0 image.png.p2p'
Node 4 listening UDP in 127.0.0.1:6004.
0 image.png.p2p
```

- É possível observar que o nó 0, ao buscar por chunks de *image.png*, recebe os chunks de 1:

```

Node 1, with 127.0.0.1:6001, received: b'{"TYPE_CLIENT": "SEND_FILE", "ADDRESS": ["127.0.0.1",
Node 1 received from ('127.0.0.1', 60485) type of connection: SEND_FILE
TRYING TO CONNECT TO TCP Ping 0: Node 1 is creating TCP client to ['127.0.0.1', '6000']
Node 0 received confirmation from 127.0.0.1:6001
Pings: 0, Elapsed: 0.0004475116729736328
Node 1 is trying to connect to TCP 127.0.0.1:6000
TCP Connection refused between node 1 and 127.0.0.1:6000: [Errno 111] Connection refused
Node 0 listening TCP on 127.0.0.1:6000 for 15 seconds
Node 0 TCP is waiting for connection
TRYING TO CONNECT TO TCP Ping 1: Node 1 is creating TCP client to ['127.0.0.1', '6000']
Node 1 is trying to connect to TCP 127.0.0.1:6000
Node 1 connected to TCP 127.0.0.1:6000
Node 1 will try to send chunks
Node 1 is opening file /home/raquel/Desktop/distribuida-trabalho1/src/../nodes/1/image.png.ch0
Connected TCP: 0 - 127.0.0.1:6000 received connection from ('127.0.0.1', 47170)
Node 0 TCP received connection
Node 1 is reading file /home/raquel/Desktop/distribuida-trabalho1/src/../nodes/1/image.png.ch0
Node 1 sent a chunk of 200 bytes to 127.0.0.1:6000

```

de 2:

```

Node 2 is trying to connect to TCP 127.0.0.1:6000
Node 2 connected to TCP 127.0.0.1:6000
Node 2 will try to send chunks
Connected TCP: 0 - 127.0.0.1:6000 received connection from ('127.0.0.1', 37354)
Node 2 is opening file /home/raquel/Desktop/distribuida-trabalho1/src/../nodes/2/image.png.ch1
Node 0 TCP received connection
Node 2 is reading file /home/raquel/Desktop/distribuida-trabalho1/src/../nodes/2/image.png.ch1
Node 2 sent a chunk of 200 bytes to 127.0.0.1:6000
Chunk IMAGE.PNG.CH1 6.67% transferred.

```

de 3:

```

TCP Connection refused between node 3 and 127.0.0.1:6000: [Errno 111] Connection refused
Node 0 listening TCP on 127.0.0.1:6000 for 15 seconds
Node 0 TCP is waiting for connection
TRYING TO CONNECT TO TCP Ping 1: Node 3 is creating TCP client to ['127.0.0.1', '6000']
Node 3 is trying to connect to TCP 127.0.0.1:6000
Node 3 connected to TCP 127.0.0.1:6000
Node 3 will try to send chunks
Node 3 is opening file /home/raquel/Desktop/distribuida-trabalho1/src/../nodes/3/image.png.ch2
Connected TCP: 0 - 127.0.0.1:6000 received connection from ('127.0.0.1', 60270)
Node 3 is reading file /home/raquel/Desktop/distribuida-trabalho1/src/../nodes/3/image.png.ch2
Node 3 sent a chunk of 200 bytes to 127.0.0.1:6000
Chunk IMAGE.PNG.CH2 6.67% transferred.
Node 0 TCP received connection

```

e de 4:

```

TRYING TO CONNECT TO TCP Ping 0: Node 4 is creating TCP client to ['127.0.0.1', '6000']
Node 4 is trying to connect to TCP 127.0.0.1:6000
Node 0 received confirmation from 127.0.0.1:6004
Pings: 0, Elapsed: 0.0009438991546630859
TCP Connection refused between node 4 and 127.0.0.1:6000: [Errno 111] Connection refused
TRYING TO CONNECT TO TCP Ping 1: Node 4 is creating TCP client to ['127.0.0.1', '6000']
Node 4 is trying to connect to TCP 127.0.0.1:6000
Node 0 listening TCP on 127.0.0.1:6000 for 15 seconds
Node 0 TCP is waiting for connection
Node 4 connected to TCP 127.0.0.1:6000
Node 4 will try to send chunks
Node 4 is opening file /home/raquel/Desktop/distribuida-trabalho1/src/../nodes/4/image.png
Connected TCP: 0 - 127.0.0.1:6000 received connection from ('127.0.0.1', 36374)
Node 0 TCP received connection
Node 4 is reading file /home/raquel/Desktop/distribuida-trabalho1/src/../nodes/4/image.png
Node 4 sent a chunk of 200 bytes to 127.0.0.1:6000
Chunk IMAGE.PNG.CH3 9.50% transferred.
Node 4 sent a chunk of 200 bytes to 127.0.0.1:6000
Chunk IMAGE.PNG.CH3 18.99% transferred.
Node 4 sent a chunk of 200 bytes to 127.0.0.1:6000

```

Exemplo 2:

- Estado inicial:

```

config.txt
config >
1 0: 127.0.0.1, 6000, 200
2 1: 127.0.0.1, 6001, 200
3 2: 127.0.0.1, 6002, 200
4 3: 127.0.0.1, 6003, 200
5 4: 127.0.0.1, 6004, 200
6

```

```

nodes
├── 0
│   ├── image.png
│   ├── image.png.ch0
│   ├── image.png.ch1
│   ├── image.png.ch2
│   └── image.png.ch3
├── 1
│   └── image.png.ch0
├── 2
│   └── image.png.ch1
├── 3
│   └── image.png.ch2
└── 4
    └── image.png.ch3

```

- Parâmetros:

Nó 4 quer receber image.png

```
4 image.png.p2p
TRYING TO SEND UDP P
```

- Resultados:

Recebe os chunks do nó 0 e nó 3:

```
TCP Connection refused between node 0 and 127.0.0.1:6004
TRYING TO CONNECT TO TCP Ping 1: Node 0 is creating TCP
Node 0 is trying to connect to TCP 127.0.0.1:6004
Node 4 listening TCP on 127.0.0.1:6004 for 15 seconds
Node 4 TCP is waiting for connection
Node 0 connected to TCP 127.0.0.1:6004
Node 0 will try to send chunks
Node 0 is opening file /home/raquel/Desktop/distribuida
Connected TCP: 4 - 127.0.0.1:6004 received connection f
Node 0 is reading file /home/raquel/Desktop/distribuida
Node 4 TCP received connection
Node 0 sent a chunk of 200 bytes to 127.0.0.1:6004
Chunk IMAGE.PNG.CH0 6.67% transferred.
Node 0 sent a chunk of 200 bytes to 127.0.0.1:6004
Chunk IMAGE.PNG.CH0 13.33% transferred.
Node 0 sent a chunk of 200 bytes to 127.0.0.1:6004
Chunk IMAGE.PNG.CH0 20.00% transferred.
```

```
Node 0 is trying to connect to TCP 127.0.0.1:6004
TCP Connection refused between node 0 and 127.0.0.1:6004
TRYING TO CONNECT TO TCP Ping 1: Node 0 is creating TCP
Node 0 is trying to connect to TCP 127.0.0.1:6004
Node 4 listening TCP on 127.0.0.1:6004 for 15 seconds
Node 4 TCP is waiting for connection
Node 0 connected to TCP 127.0.0.1:6004
Node 0 will try to send chunks
Node 0 is opening file /home/raquel/Desktop/distribuida
Connected TCP: 4 - 127.0.0.1:6004 received connection f
Node 4 TCP received connection
Node 0 is reading file /home/raquel/Desktop/distribuida
Node 0 sent a chunk of 200 bytes to 127.0.0.1:6004
Chunk IMAGE.PNG.CH1 6.67% transferred.
Node 0 sent a chunk of 200 bytes to 127.0.0.1:6004
Chunk IMAGE.PNG.CH1 13.33% transferred.
Node 0 sent a chunk of 200 bytes to 127.0.0.1:6004
Chunk IMAGE.PNG.CH1 20.00% transferred.
REQUEST TIMED OUT FOR CLIENT 1 - connection refused
```

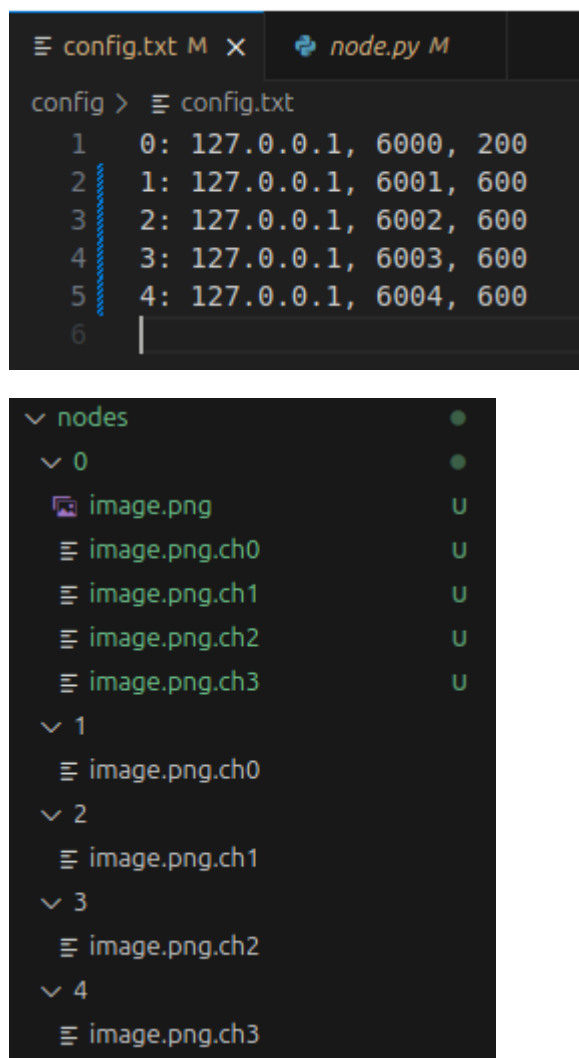
```

Pings: 0, Elapsed: 0.0004820823669433594
Node 3 is trying to connect to TCP 127.0.0.1:6004
TCP Connection refused between node 3 and 127.0.0.1:6004: [Errno 111] Connection refused
Node 4 listening TCP on 127.0.0.1:6004 for 15 seconds
Node 4 TCP is waiting for connection
TRYING TO CONNECT TO TCP Ping 1: Node 3 is creating TCP client to ['127.0.0.1', '6004']
Node 3 is trying to connect to TCP 127.0.0.1:6004
Node 3 connected to TCP 127.0.0.1:6004
Node 3 will try to send chunks
Node 3 is opening file /home/raquel/Desktop/distribuida-trabalho1/src/../nodes/3/image.png.ch2
Connected TCP: 4 - 127.0.0.1:6004 received connection from ('127.0.0.1', 46052)
Node 3 is reading file /home/raquel/Desktop/distribuida-trabalho1/src/../nodes/3/image.png.ch2
Node 4 TCP received connection
Node 3 sent a chunk of 200 bytes to 127.0.0.1:6004
Chunk IMAGE.PNG.CH2 6.67% transferred.
TRYING TO SEND UDP Ping 0: Host 0 - 127.0.0.1:6000 reading message {"type": "client", "found": false}

```

### Exemplo 3:

- Estado inicial:



The screenshot shows a code editor with two tabs: `config.txt` and `node.py`. The `config.txt` tab is active, displaying a configuration file with the following content:

```

config > config.txt
1 0: 127.0.0.1, 6000, 200
2 1: 127.0.0.1, 6001, 600
3 2: 127.0.0.1, 6002, 600
4 3: 127.0.0.1, 6003, 600
5 4: 127.0.0.1, 6004, 600
6

```

Below the code editor, a file explorer shows a directory structure:

- nodes
  - 0
    - image.png
    - image.png.ch0
    - image.png.ch1
    - image.png.ch2
    - image.png.ch3
  - 1
    - image.png.ch0
  - 2
    - image.png.ch1
  - 3
    - image.png.ch2
  - 4
    - image.png.ch3

- Parâmetros:

Nó 4 quer receber image.png

```
4 image.png.p2p
```

- Resultados:

Recebe os chunks do nó 1, nó 2, e nó 3:

Nó 1 manda .ch0:

```
chunk IMAGE.PNG.CH0 60.00% transferred.  
Node 1 is trying to connect to TCP 127.0.0.1:6004  
Node 1 connected to TCP 127.0.0.1:6004  
Node 1 will try to send chunks  
Node 1 is opening file /home/raquel/Desktop/distribuida-trabalho1/src/../nodes/1/image.png.ch0  
Node 1 is reading file /home/raquel/Desktop/distribuida-trabalho1/src/../nodes/1/image.png.ch0  
Node 1 sent a chunk of 600 bytes to 127.0.0.1:6004
```

Nó 2 manda .ch1:

```
Node 2 sent a chunk of 600 bytes to 127.0.0.1:6004  
Chunk IMAGE.PNG.CH1 60.00% transferred.
```

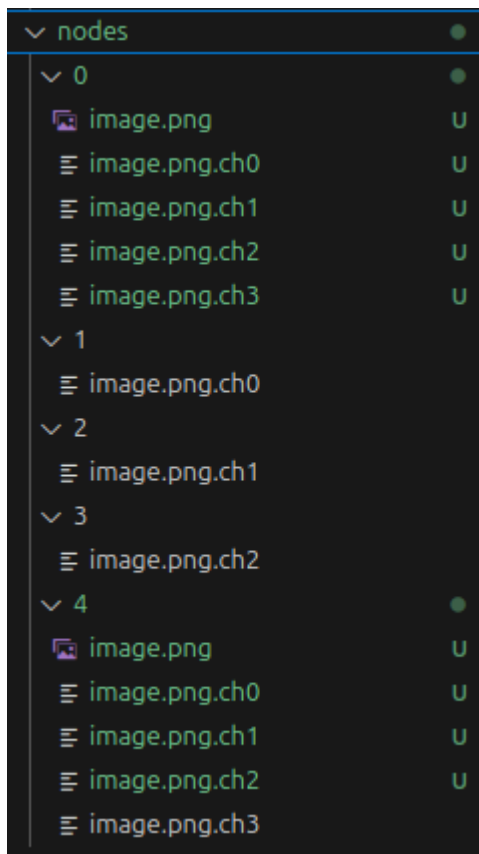
Nó 3 manda .ch2:

```
Node 3 sent a chunk of 600 bytes to 127.0.0.1:6004  
Chunk IMAGE.PNG.CH2 100.00% transferred.
```

Exemplo 3:

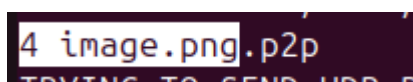
- Estado inicial:

```
config.txt M x node.py M  
config > config.txt  
1 0: 127.0.0.1, 6000, 200  
2 1: 127.0.0.1, 6001, 600  
3 2: 127.0.0.1, 6002, 600  
4 3: 127.0.0.1, 6003, 600  
5 4: 127.0.0.1, 6004, 600  
6
```



- Parâmetros:

Nó 4 quer receber image.png



- Resultados:

Como já tem os chunks, não recebe:

```
Exemplo: se quero começar a busca pelo nó 0, e o arquivo .p2p é o image.png
PERCENTAGE OF THE FILE ALREADY TRANSFERRED: 100.0%
SUCCESS: Found all chunks!
Merging files
Merging file image.png.ch0
Merging file image.png.ch1
Merging file image.png.ch2
Merging file image.png.ch3
```

## 4. Conclusões

O programa comporta-se corretamente. Isso é conferido pelos exemplos 1, 2, 3 e 4:

- No exemplo 1, todos os nós possuem taxas de transferência iguais, cada chunk está em apenas um nó, e todos os chunks estão em nós diferentes. Se o usuário definir o nó 0 como o nó que começará a busca, o nó 0 termina com todos os chunks, e a imagem é gerada;
- No exemplo 2, todos os nós possuem taxas de transferência iguais, e, além dos nós 1, 2, e 3, cada um possuir um chunk, o nó 0 possui todos os chunks. Se o usuário definir o nó 4 como o nó que começará a busca, ele recebe todos os chunks, e recebe os chunks dos nós 3 e 0;
- No exemplo 3, os nós 1, 2 e 3 possuem taxas de transferência maiores, 600, e o nó 0 possui taxa de transferência menor, 200. Também, além dos nós 1, 2, e 3, cada um possuir um chunk, o nó 0 possui todos os chunks. Se o usuário definir o nó 4 como o nó que começará a busca, ele recebe todos os chunks, mas recebe os chunks dos nós 1, 2 e 3, confirmando que receberá dos nós com taxa de transferência maiores;
- No exemplo 4, os nós 1, 2 e 3 possuem taxas de transferência maiores, 600, e o nó 0 possui taxa de transferência menor, 200. Também, além dos nós 1, 2, e 3, cada um possuir um chunk, o nó 0 possui todos os chunks **e o nó 4 possui todos os chunks**. Como o nó 4 já possui todos os chunks, ele não recebe chunks de nenhum outro nó, e já termina a execução.