Universidade do Minho
Departamento de Informática

Comunicações por Computador

Trabalho Prático II

Realizado por:
José Rodrigues (a100692)
João Coelho (a100596)
Duarte Araújo (a100750)

December 18, 2023
Ano Letivo 2023/24

Abstract. Neste trabalho foi desenvolvido um serviço de partilha de ficheiros *peer-to-peer* que utiliza dois protocolos, criados pelo grupo, um baseado em *TCP* e outro em *UDP*. Estes protocolos são responsáveis por garantir uma comunicação eficiente e rápida entre os diversos elementos da rede, o *FS_Tracker* e os *FS_Nodes*. Assim sendo, o *FS_Tracker* mantém uma conexão *TCP*, utilizando um *socket*, com cada *FS_Node* e, por sua vez, sempre que requisitado, os nodos podem iniciar a transferência de um ficheiro utilizando o protocolo *UDP* referido anteriormente. Nestes casos, também é necessário o uso de um *socket UDP* nos nodos. Durante a transferência é necessário um algoritmo que decida a quem é que será pedido o conteúdo do ficheiro a ser transferido e esse algoritmo também foi desenvolvido pelo grupo. Por fim, o projeto foi ainda adaptado de forma a utilizar plataformas *DNS*.

Keywords: Transfência · TCP · UDP · Algoritmo.

1 Introdução

Na âmbito da cadeira **Comunicações por Computador** foi-nos pedido que desenvolvêssemos um serviço de partilha de ficheiros *peer-to-peer* de alto desempenho. Para além disso, ainda nos foi indicado que deveríamos utilizar um protocolo de comunicação com base em *TCP* e um protocolo especializado em transferência baseado em *UDP*, criados exclusivamente por nós para o âmbito deste trabalho. Por fim, com o objetivo de melhorar o desempenho da transferência de ficheiros, fomos ainda informados que os ficheiros deveriam estar divididos em diversos *chunks* para que pudessem existir *chunks* do mesmo ficheiro em diversos nodos da arquitetura. Assim, se um nodo quisesse um certo ficheiro, teria de decidir a quem pedir os *chunks* que o constituem, sendo que a solução encontrada deveria ser obtida a partir de um algoritmo de decisão também desenvolvido por nós. É importante referir que este projeto foi desenvolvido utilizando *python*.

2 Arquitetura da Solução

A arquitetura final do nosso trabalho envolve a existência de um servidor **FS_Tracker** que está conectado a diversos clientes, denominados **FS_Node**, que, por sua vez, também ocupam a posição de servidores para os outros clientes da rede. É importante referir que o *FS_Tracker* tem um *socket TCP* principal para aceitar *FS_Nodes* e, sempre que aceita um, cria um *Socket TCP* exclusivo para o *FS_Node* aceitado. Para além disso, cada *FS_Node* tem também um *socket TCP* para o *FS_Tracker* e um *socket UDP* que, quando necessário, será utilizado para a transferência de ficheiros. De seguida, a transferência de ficheiros pode ser iniciada por qualquer *FS_Node* da rede, a pedido do utilizador, que deve indicar o nome do ficheiro em que está interessado. Assim, ao receber uma indicação de que um nodo pretende iniciar o processo de transferência, o *FS_Tracker* devolve

ao nodo uma lista dos *nodes* que possuem blocos do ficheiro. O algoritmo previamente referido decide então a que *nodes* pedir os blocos que necessita podendo assim ser iniciada a transferência do ficheiro. Assim, durante a transferência de um ficheiro, a arquitetura do nosso projeto tem a seguinte forma:

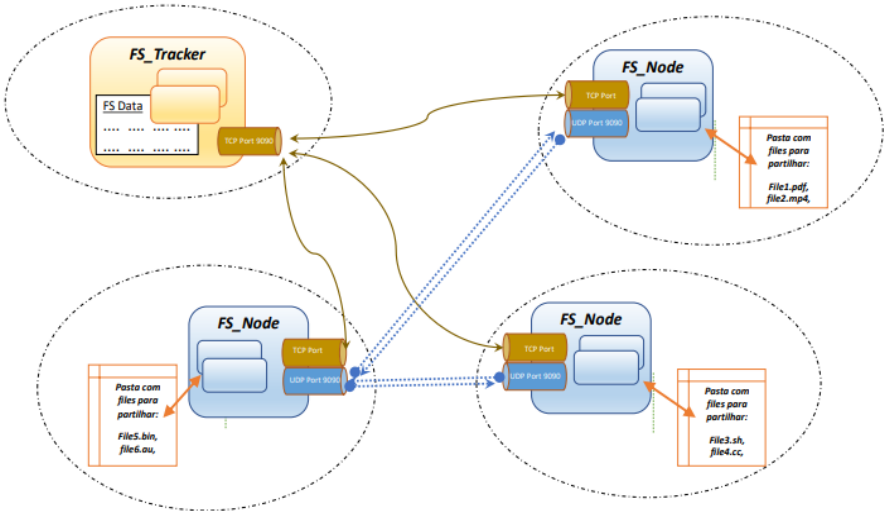
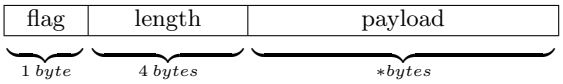


Fig. 1: Arquitetura do enunciado

3 TCP Protocol

O protocolo TCP definido para estabelecer a comunicação entre os nodos e o tracker exige que as mensagens enviadas sejam compostas por uma **flag**, o **tamanho do payload**, e, por fim, o próprio **payload**, que é o conjunto de dados a serem enviados. As diferentes *flags* são responsáveis por sinalizar o tipo da mensagem ao seu recetor. Estas podem ser:

1. Storage - Enviada do nodo para o *tracker*, com o intuito de lhe passar a informação dos ficheiros que o nodo possui, mais especificamente, os seus nomes, o número de *chunks* em que podem ser divididos, e um *array* com os *hash codes* desses *chunks*.
2. Update - Mensagem enviada pelo nodo, que informa o *tracker* da receção de um novo *chunk* de um ficheiro. A *payload* é composta pela informação do *chunk*.
3. Order - O nodo pede ao *tracker* que este lhe diga quais são os nodos que possuem o ficheiro indicado na *payload*.
4. Ship - Resposta do *tracker* à Order. Envia os nomes dos nodos com *chunks* do ficheiro, e a lista dos respetivos *chunks* de cada um.



Formato da mensagem

Assim, com o protocolo introduzido, é possível demonstrar uma simples interação entre o *FS_Tracker* e um *FS_Node*:

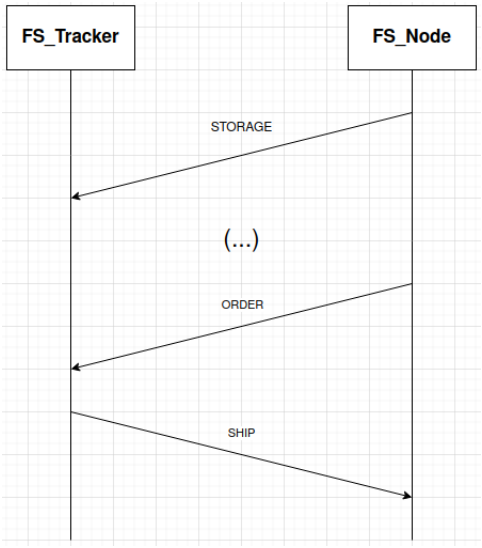


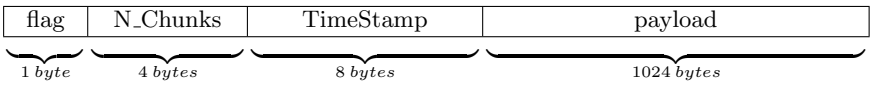
Fig. 2: Diagrama de sequência da interação TCP Tracker/Node

4 UDP Protocol

O protocolo UDP que foi desenvolvido tem como objetivo assegurar a transferência dos ficheiros entre os nodos da topologia. As mensagens que seguirem este protocolo deverão ter uma **flag**, um indicador do número do *chunk* a ser transferido (**N_Chunk**), do momento em que o pedido de transferência foi realizado (**TimeStamp**), e a **payload**, que pode ser o nome do ficheiro requirido, ou o *chunk* indicado do ficheiro em questão. As messagens podem estar identificadas com as seguintes *flags*:

1. Order - Mensagem de um nodo para outro, em que a *payload* é o nome do ficheiro requisitado. O campo *N_Chunk* indica o *chunk* a ser transferido pelo recetor.

2. Data - Resposta a uma mensagem do tipo *Order*. Todos os campos da *Order* são reenviados, à exceção da *payload*, que passa a conter o *chunk* do ficheiro.



Formato da mensagem

Assim, com o protocolo UDP explicado, é possível demonstrar uma simples interação que decorre entre o *FS_Tracker* e os *FS_Nodes* durante a transferência de um ficheiro.

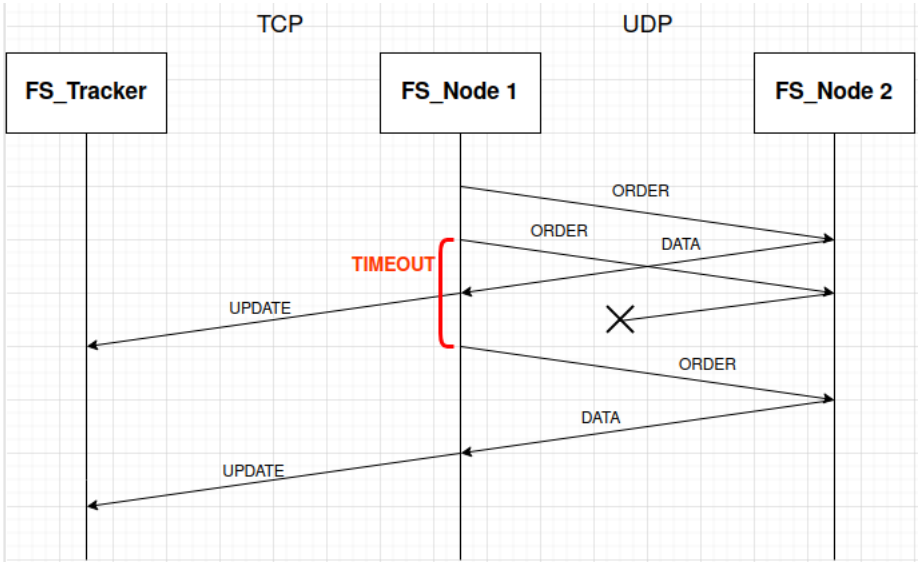


Fig. 3: Diagrama de sequência da interação UDP Tracker/Node

5 Implementação

Esta secção destina-se a descrever com mais pormenor os detalhes da implementação desenvolvida. Desta forma, iremos retratar o código *tracker* bem como dos nodos, passando pelas estruturas usadas para guardar a informação, as estratégias utilizadas para enviar e receber os dados nos protocolos especificados anteriormente e outros detalhes que se mostrem relevantes. Para além disso, ainda vai ser descrito como o mecanismo de *timeout* e o algoritmo de escolha

de nodos foram implementados. Todas as comunicações entre *sockets* foram realizadas com recurso à biblioteca *socket*[1] e o manuseamento de *threads* com recurso à biblioteca *threading*[2].

5.1 FS_Tracker

Estruturas O servidor principal possui dois *Safe Maps* (5.3), estrutura que se baseia em dicionários e que irá ser descrita mais à frente, **nodes** e **hashes**. O primeiro tem como chaves os nomes do ficheiros que existem na rede. Para cada chave está associado outro *Safe Map* que possui como chaves os nomes dos nodos que possuem blocos do ficheiro em causa. O valor para cada chave deste último dicionário é uma lista de números que referenciam o bloco que possuem.

A segunda estrutura tem como propósito armazenar as *hashes codes* para cada bloco do ficheiro. Como se pode inferir, a chave para este dicionário é o nome do ficheiro e o valor é um dicionário, desta vez nativo da linguagem *python*. A chave deste dicionário é o número do bloco e o valor é a *hash code* correspondente.

Funcionamento O código do servidor é iniciado no terminal, sendo passado como argumento a porta em que o *socket TCP* irá ser aberto. Após as estruturas serem sinalizadas, o servidor fica à espera de receber conexões de nodos, sendo que, no caso de uma eventual conexão, é criado um *socket TCP* exclusivo para este cliente. Logo após uma nova conexão, o servidor cria uma *thread* dedicada a tratar deste cliente, libertando o *socket* principal para receber novos pedidos de conexão. Esta *thread* fica também à espera de mensagens enviadas pelo nodo e cada mensagem recebida será encaminhada para uma função específica consoante a *flag* da mesma.

No caso da mensagem ser do tipo *Storage* ou *Update* o servidor irá apenas ler e interpretar o *payload* pelo formato enviado pelo nodo, obtendo assim a informação pretendida que é guardada nas estruturas referidas anteriormente. O formato do *payload* destas mensagens será melhor descrito quando for apresentado o funcionamento dos nodos.

Para além destas, existe ainda a possibilidade de receber uma mensagem do tipo *Order*. O *payload* destas mensagens corresponde ao nome do ficheiro pretendido, sendo que, após o mesmo ter sido decifrado, o servidor acede à estrutura **nodes** organizando a informação num dicionário de maneira que os nodos estejam agrupados pelos números de blocos que possuem. De seguida, esta informação é codificada em bytes junto com todas as *hashes* de cada bloco segundo a seguinte forma:

$$\{1 : [node1, node2, (...)], 2 : [node1, node3, (...)], (...)\} + [hash1, hash2, (...)] \Leftrightarrow$$
$$\Leftrightarrow b'1node1\backslash 0node2\ 2node1\backslash 0node3\ hash1hash2'$$

Desta forma, é enviado ao nodo uma mensagem com o tipo *SHIP* e um *payload* da forma que foi descrito acima.

[1] `https://docs.python.org/3/library/socket.html`
[2] `https://docs.python.org/3/library/threading.html`

5.2 FS_node

Estruturas Os nodos fazem uso dos dois protocolos. No entanto, apenas existem estruturas para armazenar informações referente ao protocolo UDP. Com efeito, este também possui dois *Safe Maps*, ambos preenchidos na altura de iniciar a transferência de um ficheiro e são esvaziados no final da mesma. Um deles, denominado *waitingchunks*, indica os blocos que ainda não foram recebidos do ficheiro. O dicionário agrupa o número do bloco com a respetiva *hash* que irá ser comparada no momento de reçeção do mesmo. A segunda estrutura, denominada *threads_timeout*, guarda as *threads* criadas para cada *timeout*, organizadas por número de bloco. O propósito desta estrutura será melhor detalhado na secção dedicada aos *timeouts*(5.4).

Os nodos possuem também um dicionário, *nodes*, para guardar informações de performance dos nodos com quem já comunicaram. Assim sendo, neste dicionário, em que a chave é o nome do nodo, está associada uma lista de quatro elementos. Desta forma, é guardado o total de *Round Trip Time* de todas as mensagens enviadas, informação que também se encontra guardada na lista, bem como o número de mensagens recebidas, o que permite calcular o *Packet loss*. Por fim, ainda está guardado o IP do nodo para evitar *queries DNS* desnecessárias.

Inicialização O programa do nodo é iniciado com três argumentos: caminho para a pasta de ficheiros que deseja partilhar; nome da máquina onde o *FS_tracker* está a ser executado; porta para se conectar ao servidor.

Após ser efetuada a conexão com o servidor, o nodo abre também um *socket* UDP que se comporta como um servidor. Este *socket* tem como propósito receber pedidos de transferência de outros nodos.

Funcionamento Após a inicialização, a primeira coisa a se fazer é enviar a informação dos ficheiros da pasta fornecida ao programa. Desta forma, para cada ficheiro da pasta, é calculado o número de blocos e as *hashes* de cada um com recurso à biblicoteca *hashlib*[3]. Assim sendo, é enviada a mensagem de tipo *Storage* ao servidor. O *payload* desta mensagem tem a informação de cada ficheiro, codificado da seguinte forma:

$$Nome\_Ficheiro + N\_Blocos + [hashs] \Leftrightarrow$$
$$\Leftrightarrow b'Nome\_Ficheiro\backslash tN\_Blocos\backslash thash1hash2(\ldots)'$$

O cliente pode assim começar a fazer transferências de ficheiros com o comando *"order"*, junto com o nome do ficheiro. Após ter enviado a mensagem TCP ao servidor a efetuar o pedido, o nodo recebe a resposta do mesmo, que contém os blocos e respetivos nodos, para além das *hashes*. Neste momento, a estrutura *waitingchunks* é preenchida e o algoritmo (5.5) é corrido sobre a informação recebida, retornando um dicionário em que a chave é o nodo a quem se vai pedir os blocos que estão no valor da chave, como lista. Para cada nodo

[3] `https://docs.python.org/3/library/hashlib.html`

é aberto um *socket UDP* com o objetivo de paralelizar o envio de pedidos de blocos. No final de todas as mensagens referentes ao nodo em questão serem enviadas, este *socket* é fechado. Importante de referir que para cada mensagem enviada é atualizado esse mesmo facto no dicionário *nodes*.

Como referido anteriormente, paralelamente a este processo anterior, um *socket UDP* está à espera de receber mensagens. Desta forma, é possível receber blocos do ficheiro mesmo não tendo ainda enviado todos os pedidos. Sempre que um bloco é recebido, marcado pela *flag Data*, é calculada a sua *hash*, que é comparada com a recebida pelo servidor, verificando desta maneira se o bloco chegou íntegro. Se o bloco estiver intacto, é então aberto o ficheiro e escrito o conteúdo na posição correta. Para além disso, a entrada deste bloco na estrutura *waitingchunks* é apagada, sinalizando que o bloco já foi recebido. De seguida, as informações referentes ao nodo, RTT e número de mensagens recebidas, são atualizadas. Em adição, o nodo ainda informa o servidor sobre este novo bloco recebido, enviando uma mensagem TCP com a *flag Update*, contendo no *payload* apenas o número do bloco junto com o nome do ficheiro.

Iremos agora descrever o lado do nodo que recebe o pedido do bloco. Nesta vertente, o seu próprio servidor *UDP* irá receber a mensagem com a *flag Order*. Desta forma, ele irá descodificar o nome do ficheiro que se encontra no conteúdo da mensagem, bem como o número do bloco e, de seguida, irá aceder ao ficheiro pedido, ler os *bytes* correspondentes ao bloco e irá enviar a resposta ao nodo com o conteúdo do ficheiro. O nodo que recebe o pedido não altera o *timestamp* recebido, limitando-se a reenviar o valor na mensagem de requisição do bloco.

5.3 Safe Map

Safe Map é uma estrutura desenvolvida para trabalhar com dicionários em ambientes de múltiplas *threads*. Desta forma, para prevenir situações de corrida no acesso ás informações dos dicionários, esta estrutura integra o mecanismo de *locks* no acesso aos mesmos. Assim sendo, quando, por fora, se acede a um destes dicionários, não é necessário ter atenção a situações de corrida visto que a própria estrutura já trata desses casos.

5.4 Timeout

Sempre que uma requisição de um bloco é enviada um *timeout* é criado em forma de classe, à qual é adicionada ao dicionário *threads_timeouts*. Esta classe possui um evento associado, que é sinalizado no momento em que o bloco é recebido. Neste caso, o *timeout* é cancelado, no entanto, no caso de passar $0.5s$ sem que o evento seja sinalizado, é criado um novo *socket UDP* para reenviar o pedido do bloco. Resumindo, uma mensagem será reenviada caso tenha passado o tempo referido sem receber o bloco ou no caso de receber, mas com erros.

5.5 Algoritmo

Por fim, foi necessário desenvolver um algoritmo para escolher a melhor distribuição dos blocos do ficheiro pelos nodos. Como referido anteriormente, este

algoritmo recebe um dicionário em que as chaves são os números dos blocos e os valores são a lista de nodos que os contém. Após o algoritmo correr, o resultado tem a forma de um dicionário cujas chaves são os nodos, que estão associados à lista de blocos que lhes foram atribuídos.

O algoritmo tem como objetivo geral conhecer e obter informações de nodos da rede. Para além disso, também prioriza o paralelismo, entregando vários blocos a diferentes nodos em lugar de pedir todos a apenas um, mesmo que este tenha uma melhor performance.

A performance de um nodo é calculada com base no seu RTT e *Packet Loss*. Para efeitos de comparar nodos utiliza-se a seguinte fórmula:

$$\frac{RTT}{RTT_{max}} \times RTT_{weight} + Packet\_Loss \times Packet\_Loss_{weight} \tag{1}$$

Em que o RTT_{max} é o maior RTT registado até ao momento e os pesos dos parâmetros são valores entre 0 e 1, escolhidos por nós. No caso foi escolhido 0.5 para os dois, equilibrando assim o peso. Com esta fórmula, quanto menor for o resultado, mais rápido é o nodo. Assim, é possível sempre decidir o melhor nodo para um determinado bloco, sendo que, em casos de empate, o bloco será atribuído ao nodo com menos blocos associados, aumentando o paralelismo.

Para além disso, o algoritmo está ainda preparado para distribuir o bloco da maneira mais eficiente possível entre dois nodos, sendo que um é mais rápido que o outro. Com efeito, ao dividir os resultados das fórmulas, mais especificamente, entre o mais lento e o mais rápido, obtemos o valor de quantas vezes é que um é mais eficiente que o outro, valor que irá ser aplicado na distribuição de blocos. Ou seja, se um nodo for x vezes mais rápido que outro nodo, o bloco será atribuído ao mais lento se, e só se:

$$N\_Blocos_{Lento} \times x < N\_Blocos_{Rápido}$$

Sendo N_Blocos o número de blocos atribuídos ao nodo em questão.

5.6 DNS

A última parte do trabalho e da implementação requeria que adaptássemos o sistema para trabalhar com *DNS* e não com *IPs* diretamente. Assim sendo, damos uso à biblioteca *socket* da linguagem, que já possui métodos para realizar as *queries DNS*. Deste modo, não foi preciso alterar muito o código desenvolvido até o momento, no entanto, ainda foi necessário configurar o *DNS*, usando o *bind9* na máquina *core*. Para isso, foi alterado o ficheiro "named.conf.default-zones" e foram criados diversos ficheiros para definir as zonas de encaminhamento e zonas reversas.

```
zone "cc" {
        type master;
        file "/etc/bind/zones/db.cc";
};

zone "1.1.10.in-addr.arpa" {
        type master;
        file "/etc/bind/zones/db.10.1.1";
};

zone "2.2.10.in-addr.arpa" {
        type master;
        file "/etc/bind/zones/db.10.2.2";
};

zone "3.3.10.in-addr.arpa" {
        type master;
        file "/etc/bind/zones/db.10.3.3";
};

zone "4.4.10.in-addr.arpa" {
        type master;
        file "/etc/bind/zones/db.10.4.4";
};
```

(a) Zonas adicionadas

```
$ORIGIN cc.
;
; BIND data file for local loopback interface
;
$TTL    604800
@       IN      SOA     Servidor1.cc. admin.cc. (
                              3         ; Serial
                         604800         ; Refresh
                          86400         ; Retry
                        2419200         ; Expire
                         604800 )       ; Negative Cache TTL
;
@       IN      NS      Servidor1.cc.
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
```

(b) Zona de encaminhamento

```
$ORIGIN 1.1.10.in-addr.arpa.
;
; BIND reverse data file for local loopback interface
;
$TTL    604800
@       IN      SOA     Servidor1.cc. admin.cc. (
                              3         ; Serial
                         604800         ; Refresh
                          86400         ; Retry
                        2419200         ; Expire
                         604800 )       ; Negative Cache TTL
;
@       IN      NS      Servidor1.cc.
1       IN      PTR     Portatil1
2       IN      PTR     Portatil2
```

(c) Exemplo de zona reversa

6 Testes e Resultados

Nas imagens desta secção, podem ser analisados os resultados de uma transferência de um ficheiro de 25 KB. Os envios dos *chunks* foram divididos entre os dois nodos que possuíam o ficheiro, pelo algoritmo desenvolvido para otimizar o processo. O nodo inferior esquerdo reenviou o *chunk* número 13, que não conseguiu chegar ao destino da primeira vez. No final, todos os *chunks* foram recebidos e o servidor foi notificado.

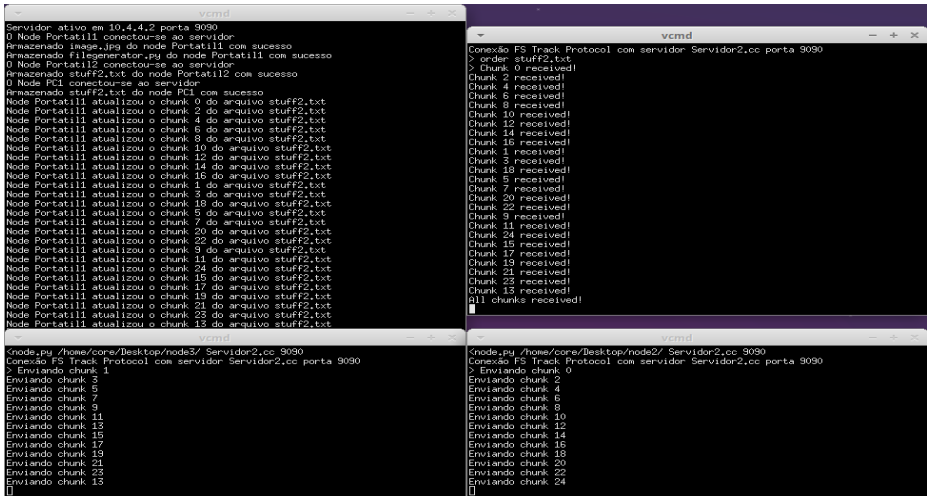


Fig. 5: Transferência de um ficheiro de 25Kb

7 Conclusões

O desenvolvimento deste projeto permitiu aprofundar o conhecimento acerca do funcionamento das redes, não só do ponto de vista prático, mas também do teórico. Para além disso, o trabalho final produzido encontra-se bastante completo e respeita todos os tópicos que lhe são exigidos e, na nossa opinião, encontra-se bastante satisfatório. Para além disso, achamos que o projeto demonstra que a matéria abordada nas aulas foi entendida e, por sua vez, utilizada para o desenvolvimento do mesmo. No entanto, o trabalho podia ser melhorado em diversos aspetos, tais como, o uso de chunks dinâmicos em vez de estáticos, melhoria da *interface* do utilizador, o sistema de *timeouts* poderia ser mais eficiente, utilizar uma *thread pool* em vez de uma *thread* por *timeout*, pedir os chunks todos de uma vez, ao contrário de ir pedindo um a um, mas evitando afetar o cálculo do *RTT (Round Trip Time)* e, no geral, o algoritmo criado para a decisão dos pedidos a enviar durante a transferência podia ser melhorado, no entanto, no seu estado atual, o mesmo já está bastante completo e cumpre todos os requisitos pretendidos.