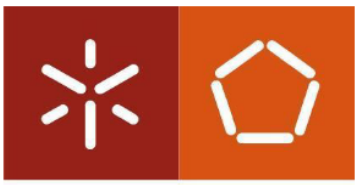


Universidade do Minho

COMUNICAÇÕES POR COMPUTADOR

TRABALHO PRÁTICO 2:
Transferência rápida e fiável de múltiplos servidores em simultâneo

Grupo nº65

a100824
Gonçalo Daniel Machado Costa

a100593
Marta Sofia Matos Castela Queirós Gonçalves

a100743
Marta Raquel da Silva Rodrigues


NoLosses
SO YOUR FILE DOESNT GET LOST

Índice

1. Proposta de Trabalho

Neste trabalho foi proposto o desenvolvimento de um serviço de envio e pedido de ficheiros *peer-to-peer* a partir do uso de dois protocolos, *TCP* e *UDP*. Estes responsáveis por garantir a comunicação entre os diversos *Nodes* e o *Tracker*. Foi também necessário o desenvolvimento de um algoritmo de decisão para tornar a troca de ficheiros rápida e pouco custosa. Ainda foi pedido que o grupo implementasse um sistema de DNS, no qual foi usado o BIND9.

2. Introdução

Em prol da disciplina de Comunicações por Computador, fomos incumbidos de criar um serviço de partilha de ficheiros peer-to-peer eficiente e rápido. Além disso, foi especificado que deveríamos empregar um protocolo de comunicação baseado em TCP e um protocolo especializado em transferência baseada em UDP, ambos desenvolvidos pelos elementos do grupo. Por fim, visando otimizar a transferência de arquivos, fomos orientados a dividir os ficheiros em vários chunks, permitindo a existência de chunks do mesmo arquivo em diferentes *nodes*. Dessa forma, se um *node* desejasse um determinado ficheiro, teria que decidir a quem solicitar os chunks que o compõem, sendo que a solução encontrada deveria ser resultado de um algoritmo de decisão.

Como ferramentas de trabalho recorremos à utilização de Python, por ser uma linguagem intuitiva de ser usada e que possui muitas bibliotecas de apoio, assim como o uso do BIND9, como proposto pelo professor nas aulas práticas.

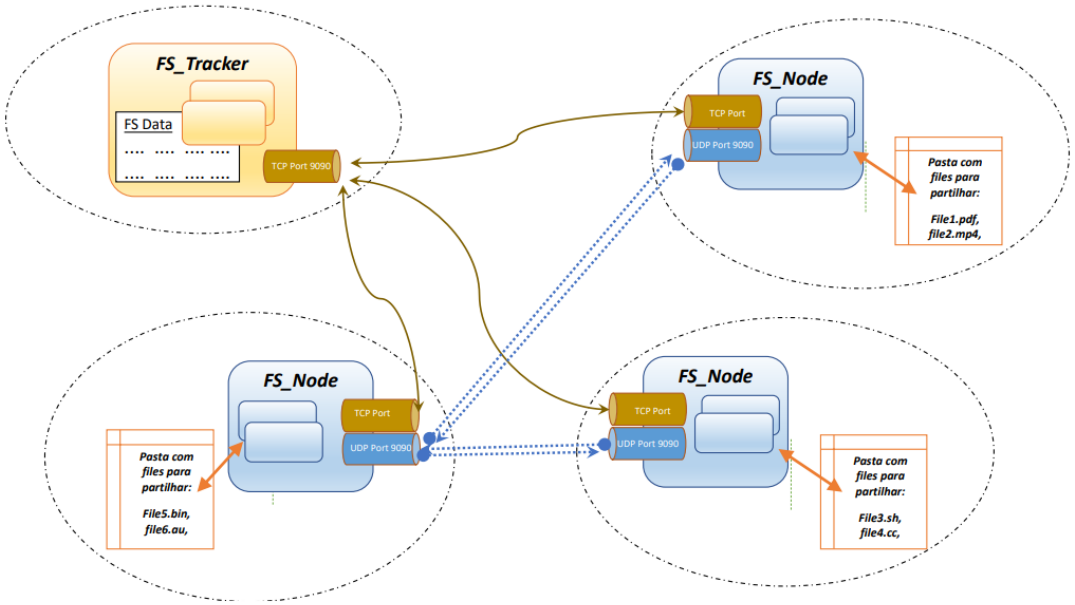


Figura 1 - Arquitetura proposta

3. Arquitetura e implementação

Para a resolução do problema proposto pelo enunciado, implementamos um servidor ***"tracker_server"*** que tem a capacidade de se conectar a diversos clientes ***"clientakanode"***. O servidor possui um *socket* TCP usado para aceitar as conexões com os clientes, após a conexão estabelecida, cria outro *socket* TCP exclusivo para esse cliente conectado, processo este que é repetido cada vez que há uma conexão com um novo cliente e essa comunicação fica reservado a uma thread permitindo assim que o programa corra de forma concorrente. Cada cliente possui também um *socket* TCP e um *socket* UDP, para a comunicação com o servidor e para a transferência de ficheiros entre clientes, respectivamente. A transferência de ficheiros acontece quando um cliente indica que está interessado num certo ficheiro, um servidor indica então quais clientes possuem chunks desse ficheiro e a partir do algoritmo de decisão são escolhidos a quais nodes serão pedidos os chunks.

3.1 Tracker_server

O servidor quando é iniciado fica à espera de se conectar com pelo menos um cliente, e como mencionado anteriormente, cria um *socket* TCP exclusivo para esse cliente e uma thread também exclusiva para lidar com o mesmo, e a mesma tratará de receber as mensagens enviadas pelo cliente e mediante a flag recebida no protocolo emparelhar a mensagem com uma função específica. No caso de a flag ser "GET" o payload irá conter o *node*, o id do ficheiro, o número de chunks desse ficheiro e a lista dos mesmos. Após isso a mensagem é codificada em bytes e enviada.

As informações ficam então armazenadas num dicionário externo que contém outro interno, seguindo a seguinte estrutura {nome ficheiro : (número de chunks, {ip: [lista de chunks]})}, o armazenamento dessa informação está responsável pela flag "GUARDA".

3.2 Clientakanode

O programa é iniciado com dois argumentos, arg[0] = caminho da pasta com que o *node* é inicializado, arg[1] = o nome da máquina da topologia que inicia o cliente. Após isso, a conexão é efetuada com o servidor e o cliente envia a informação dos ficheiros contidos na pasta inicializada e abre também, um *socket* UDP, com o propósito de receber pedidos de transferência de ficheiros.

Para cada ficheiro, o servidor irá calcular o número de chunks e armazená-lo na estrutura mencionada anteriormente.

3.3 Algoritmo de Decisão

Implementamos ainda um algoritmo que decidisse a que nodos iriam ser pedidos os chunks. Acabamos por implementar um algoritmo muito simples com uma variável "chunk_atual" que ia aumentando sempre que se realizava o pedido correspondente a esse chunk. O algoritmo percorria o dicionário as vezes necessárias para que todos os chunks

fossem pedidos a algum nodo. Caso o nodo possuísse esse chunk, era-lhe enviado o pedido e o “chunk_atual” avançava; caso contrário, o “chunk_atual” mantinha-se e verificava-se a sua existência no nodo seguinte.

Um algoritmo semelhante era aplicado algum tempo depois, equivalente à variável “timeout” calculada a partir do tamanho do ficheiro: quanto maior o tamanho, maior o tempo de timeout antes de efetuar um novo pedido de chunks. A única diferença é que para cada valor do “chunk_atual” verificava-se se este aparecia na lista de chunks recebidos e se aparecesse, avançava-se para o seguinte.

4. Protocolo TCP (FS Track Protocol)

A função principal do protocolo TCP é estabelecer a ligação entre os clientes e o servidor, tendo uma composição pré definida pelo grupo das mensagens trocadas entre eles, composição essa com a seguinte estrutura:

- 1 byte que representa uma Flag
- 4 bytes que representam a length do payload
- Payload que possui tamanho variado
- As flags mencionadas em cima têm a função de indicar ao receptor a função da mensagem que o mesmo recebe, podendo ser estas:
    - GUARDA → O cliente envia a informação de que ficheiros e chunks possui ao servidor.
    - GET → O cliente pede ao servidor que lhe indique quais outros clientes possuem o ficheiro desejado.
    - SEND → O servidor envia para o cliente as informações sobre o ficheiro solicitado.

A comunicação entre o cliente e o servidor pode ser representada através de alguns diagramas como os mostrados de seguida.

Antes de mais, como já foi referido, o FS_Tracker encontra-se à escuta e um FS_Node estabelece uma conexão com ele e depois de algumas trocas de informação o cliente pode desconectar-se, fazendo com que o servidor elimine todas as informações que possui acerca dele.

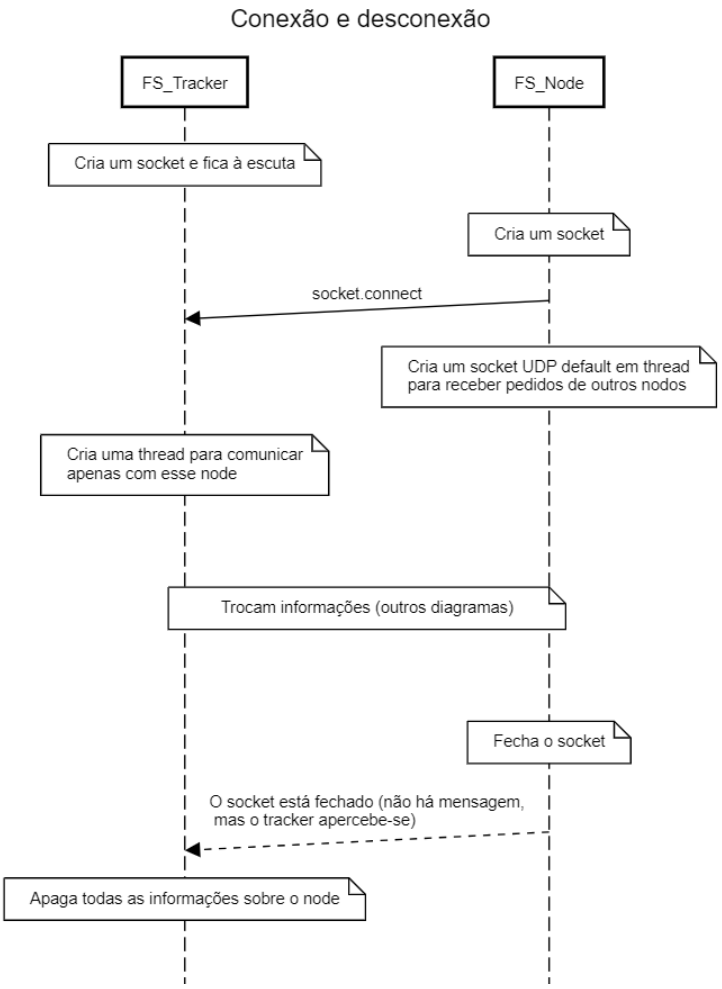


Figura 2 - Diagrama de Sequência de Conexão/Desconexão

A primeira troca de informações é automática e é feita logo após ser estabelecida a conexão. O FS_Node verifica a pasta que lhe é passada como argumento e envia informações ao servidor sobre os ficheiros nela contidos, como o seu nome e o seu tamanho. Há ainda um indicador (a que chamamos total) que indica que possui o ficheiro completo.

Já o FS_Tracker guarda todas essas informações nos respectivos dicionários (id e tamanho ficam associados ao nome do ficheiro no file_dict e os chunks que possui ficam associados ao nome do nodo e ao nome do ficheiro no node_super_dick).

O FS_Node recebe ainda informações sobre o id do ficheiro e guarda-as no seu próprio file_dict.

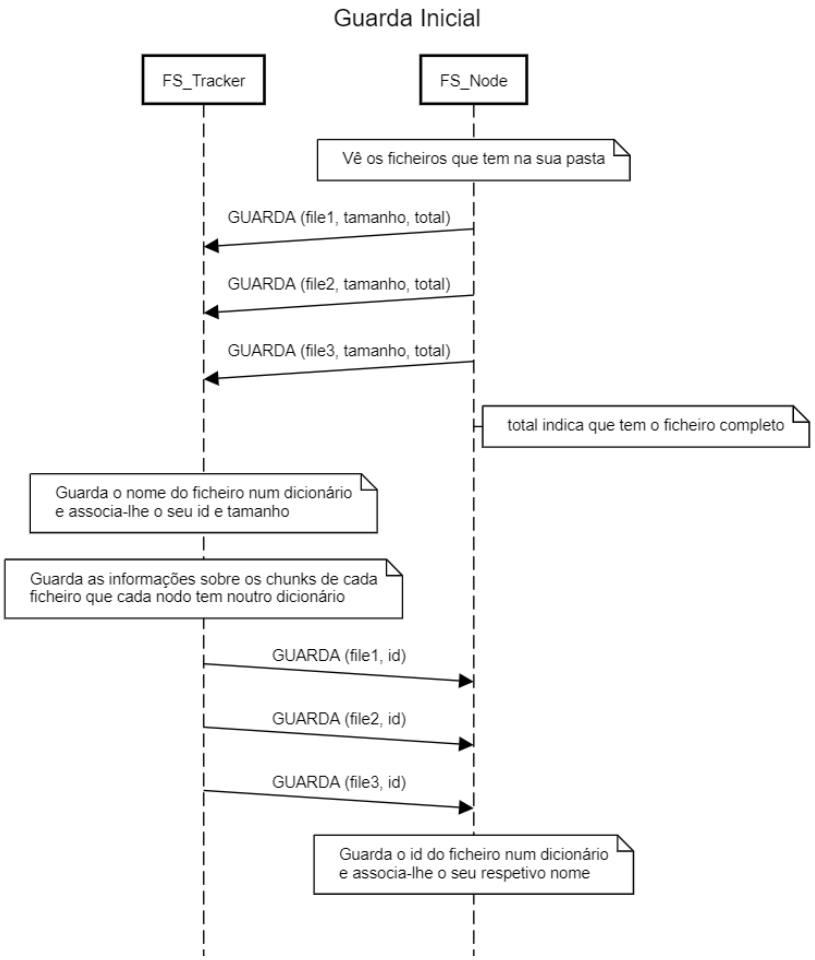


Figura 3 - Diagrama de sequência Guarda Inicial

Ao receber chunks dos ficheiros, o FS_Node informa o FS_Tracker que os possui, através de um GUARDA com o indicador parcial.

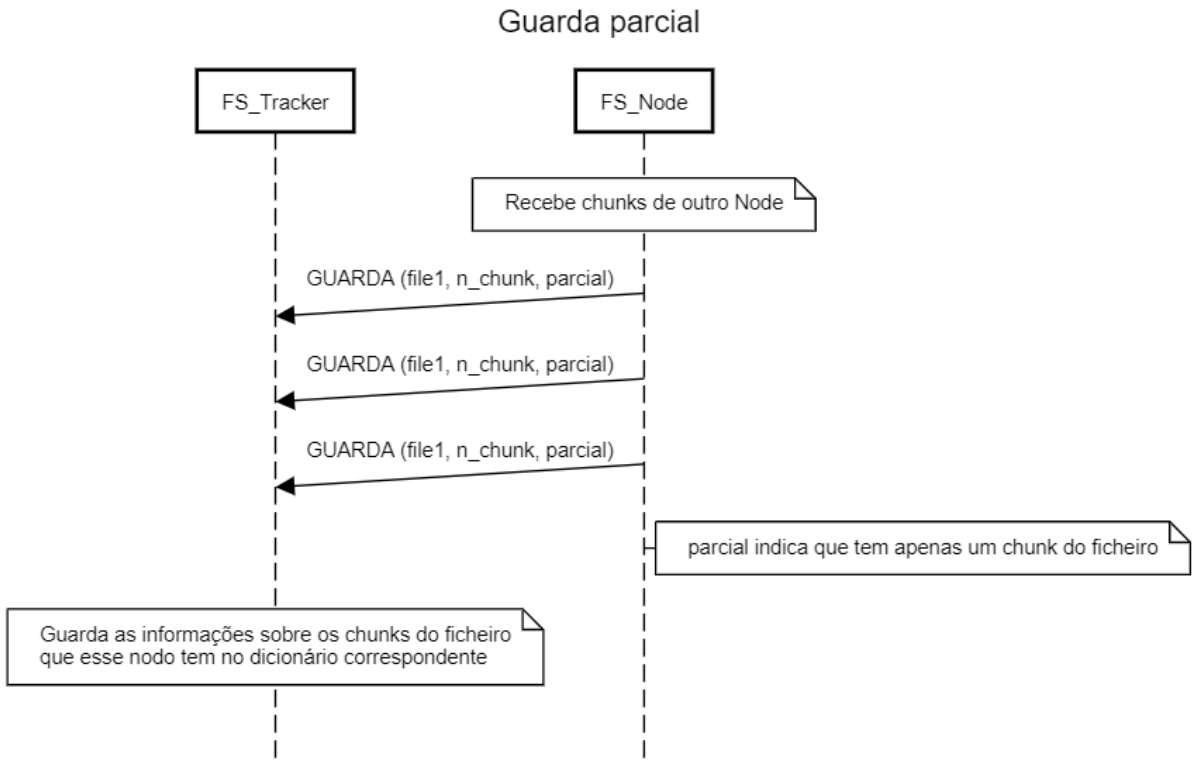


Figura 4 - Diagrama de Sequência Guarda Parcial

Além de todas estas comunicações, existe ainda a comunicação GET quando um cliente quer um ficheiro mas esta será mencionada mais à frente pois está relacionada com o protocolo UDP.

5. Protocolo UDP (Transfer Protocol)

O protocolo UDP estará encarregue por gerir a função de *Transfer Protocol,* isto é, a descarga de ficheiros por qualquer FS Node a pedido do utilizador, estabelecendo a ligação entre dois distintos clientes.

Deste modo, os datagramas UDP possuem um formato base, cuja estrutura é a seguinte:

- 4 bytes para *Checksum,* que representa o valor computado do datagrama UDP total excluindo o próprio *Checksum.*
- 4 bytes (int) que representam o ID do ficheiro que está a ser transferido.
- 4 bytes (int) que representam o Offset, que permite identificar o bloco (*chunk*) que o datagrama está a transferir, correspondente então à posição a partir da qual os dados pertencem num ficheiro.
- 4 bytes (int) para representar o tamanho do payload, ou seja, dos dados que possui
- Tamanho variado de bytes para o payload da mensagem.

É importante notar que o *Checksum* é calculado recorrendo à biblioteca **zlib,** pois, apesar de não ser a mais rápida das formas de calcular esse valor, foi a biblioteca mais acessível e eficiente encontrada para o cálculo pretendido. Mais especificamente, foi usado **zlib.crc32** que devolve um checksum de 32 bits, 4 bytes como foi mencionado anteriormente. O

Checksum será útil para a deteção de erros durante as transferências realizadas, sendo que o cálculo é realizado na criação do datagrama e verificado na análise do mesmo.

Inicialmente, foi atribuído um byte para uma flag, de valores GET ou SEND, que indicaria aos clientes o tipo de mensagem que estaria a receber. No entanto, este foi posteriormente retirado, pois, no funcionamento do protocolo *FS Transfer Protocol,* os datagramas GET seriam enviados para um *socket default* (Porta 12345) e os pedidos SEND para um *socket* criado para receber os pedidos do ficheiro. Deste modo, como as portas eram distintas, não se sentiu a necessidade de possuir uma flag, poupando assim 1 byte no protocolo.

O diagrama de seguida apresentado retrata um pouco do que foi explicado anteriormente, representando o que acontece quando um cliente pretende um determinado ficheiro, desde a comunicação com o tracker para obter as informações acerca do ficheiro e dos nodes que o possuem até aos pedidos e respostas dos outros nodes. Verifica-se então uma comunicação com o FS_Tracker através do FS Track Protocol e comunicações com os nodes através do Transfer Protocol.

Note-se ainda que as duas threads criadas no FS_Node1 correm em paralelo sendo possível pedir chunks e receber outros em simultâneo, apesar de isso não estar explícito no diagrama devido à complexidade que acrescentava.

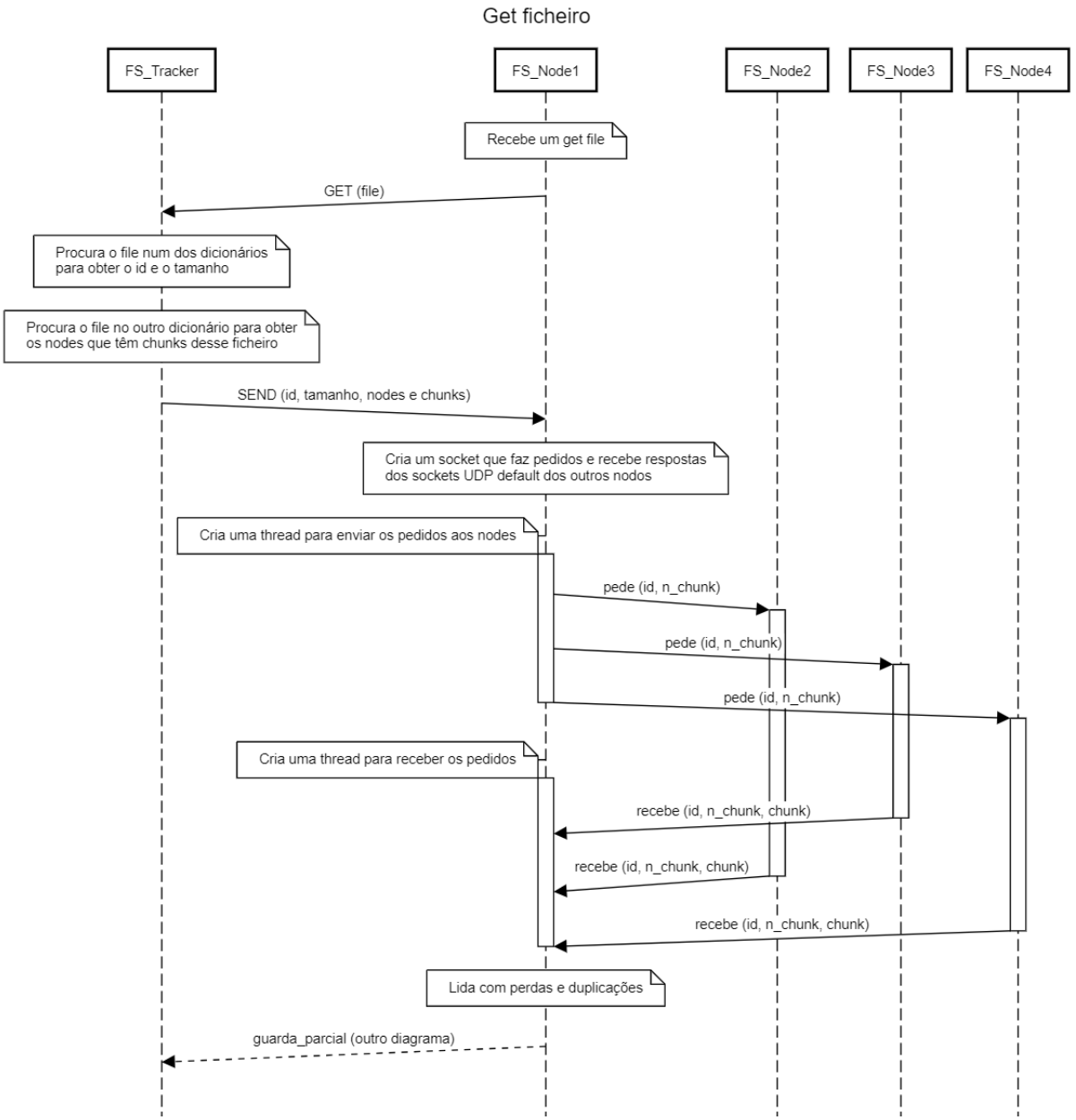


Figura 5 - Diagrama de sequência Get Ficheiro

6. DNS

Numa fase final da implementação foi desenvolvido um serviço DNS (*Domain Name System*) que permite que o servidor e os clientes passem a se identificar pelos seus nomes, em vez dos seus endereços IPv4. Para tal, foi utilizada, como ambiente de teste, a topologia previamente fornecida “CC-Topo-2023.imn” e, para configuração do serviço, o *BIND9*.

Assim, utilizamos a biblioteca *socket* que já possui métodos para realizar as *queries* DNS (**gethostbyname(hostname)** e **gethostbyaddr(ip)**) permitindo, então, que o código das funcionalidades anteriormente implementadas não fosse muito alterado.

No entanto, foi necessário alterar certas partes da implementação, como o dicionário presente no tracker, que passa a armazenar os nomes dos hosts em vez dos endereços ip e

nos nodos, para que possam resolver os nomes em endereços ip. Adicionalmente, alterou-se com os serviços de suporte para configurar o DNS, utilizando o *BIND9* na máquina *core*. Deste modo, foi alterado o ficheiro *"named.conf.default-zones"* e foi ainda criados ficheiros para definir zonas de encaminhamento (forwarding), *"cc2023.pt"* que associa um endereço ip ao nome, e zonas reversas, *"cc2023.rev"* capazes de associar um nome a um endereço ip pretendido. Todas as configurações realizadas encontram-se a seguir:

```
$ORIGIN cc.
$TTL    1d
@       IN      SOA     Servidor1.cc. admin.cc. (
                              2         ; Serial
                         604800         ; Refresh
                          86400         ; Retry
                        2419200         ; Expire
                         604800 )       ; Negative Cache TTL
; Name servers - NS records
@       IN      NS      Servidor1.cc.
@       IN      NS      Servidor2.cc.

; Define host mappings
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
Roma    IN      A       10.3.3.1
Paris   IN      A       10.3.3.2
PC1     IN      A       10.2.2.1
PC2     IN      A       10.2.2.2
~
```

Figura 6 - Zona de Encaminhamento

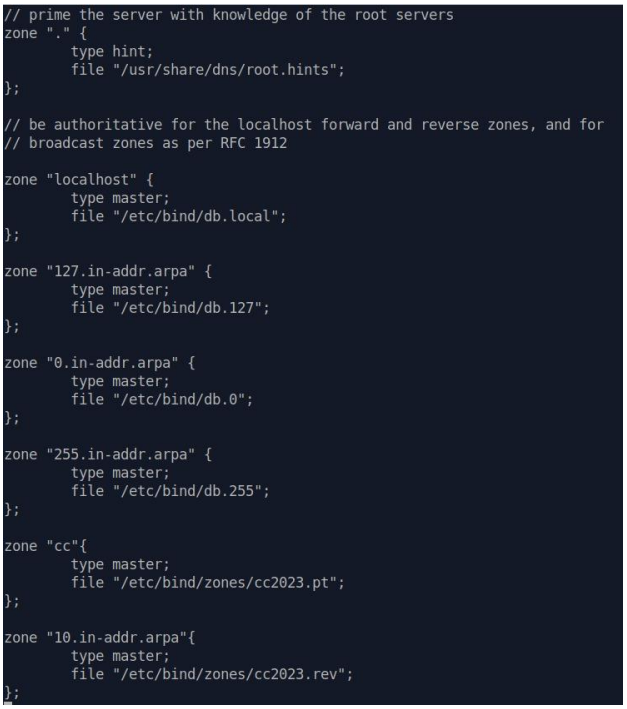
```
$ORIGIN 10.in-addr.arpa.
$TTL    604800
@       IN      SOA     Servidor1.cc. admin.cc. (
                              1         ; Serial
                         604800         ; Refresh
                          86400         ; Retry
                        2419200         ; Expire
                         604800 )       ; Negative Cache TTL
;
@       IN      NS      Servidor.cc.
1.4.4   IN      PTR     Servidor1
2.4.4   IN      PTR     Servidor2
1.1.1   IN      PTR     Portatil1
2.1.1   IN      PTR     Portatil2
1.2.2   IN      PTR     PC1
2.2.2   IN      PTR     PC2
1.3.3   IN      PTR     Roma
2.3.3   IN      PTR     Paris
~
```

Figura 7 - Zona Reversa

```
zone "cc"{
        type master;
        file "/etc/bind/zones/cc2023.pt";
};

zone "10.in-addr.arpa"{
        type master;
        file "/etc/bind/zones/cc2023.rev";
};
```

Figura 8 - Zonas Adicionais



```
// prime the server with knowledge of the root servers
zone "." {
        type hint;
        file "/usr/share/dns/root.hints";
};

// be authoritative for the localhost forward and reverse zones, and for
// broadcast zones as per RFC 1912

zone "localhost" {
        type master;
        file "/etc/bind/db.local";
};

zone "127.in-addr.arpa" {
        type master;
        file "/etc/bind/db.127";
};

zone "0.in-addr.arpa" {
        type master;
        file "/etc/bind/db.0";
};

zone "255.in-addr.arpa" {
        type master;
        file "/etc/bind/db.255";
};

zone "cc"{
        type master;
        file "/etc/bind/zones/cc2023.pt";
};

zone "10.in-addr.arpa"{
        type master;
        file "/etc/bind/zones/cc2023.rev";
};
```

Figura 9 - Resultado do ficheiro named.conf.default-zones

7. Demonstração

Nas imagens a seguir podemos ver os resultados obtidos na transferência de dois ficheiros de diferente tamanho. Como podemos ver há um cliente a fazer o pedido do ficheiro e outros nodos a enviar os chunks desse ficheiro paralelamente, além disso podemos ver que ele pede chunks em falta, sendo assim capaz de lidar com perdas de pacote.

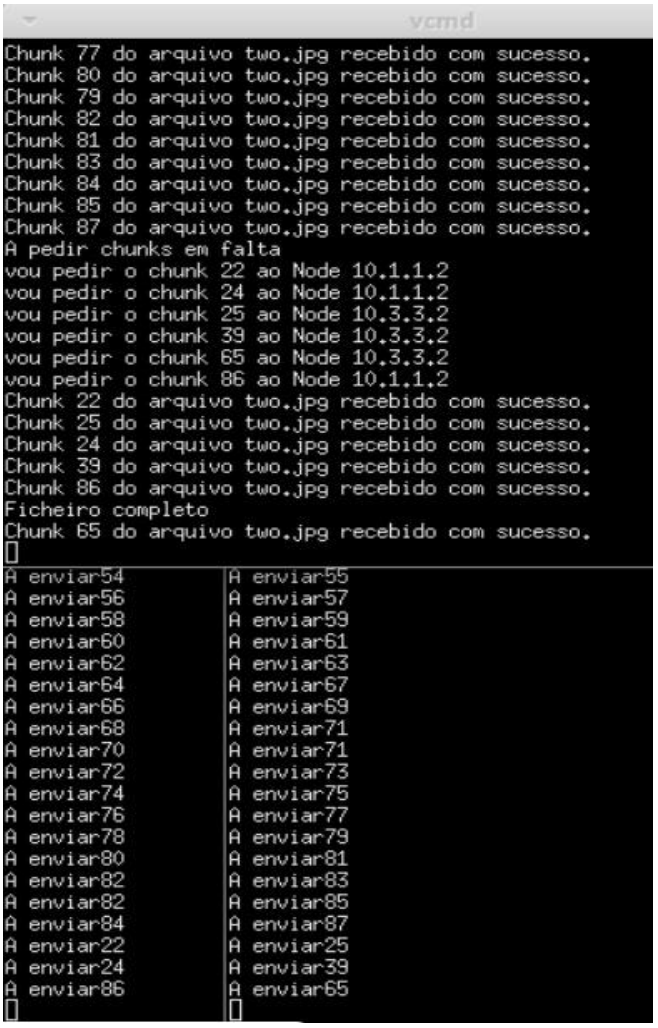


Figura 10 - Envio de um ficheiro de 84,8 KiB com resposta de 2 nodos

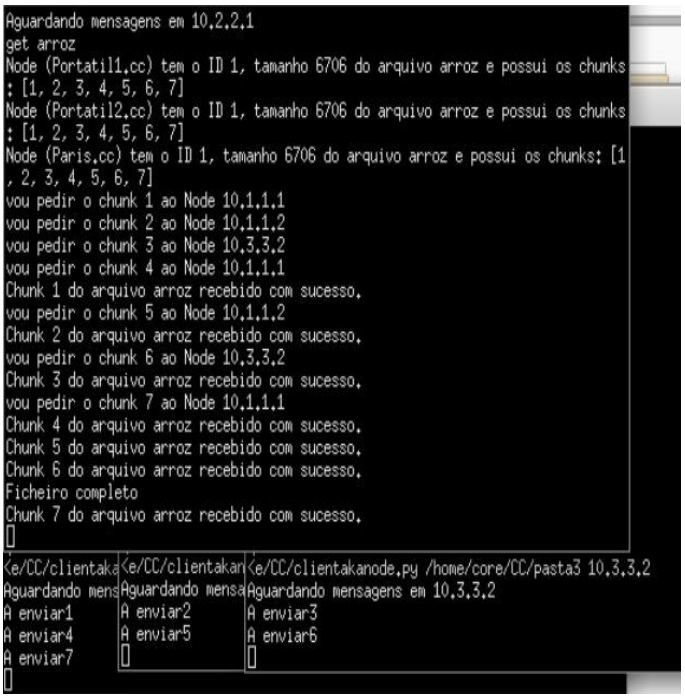


Figura 11 - Envio de um ficheiro de 6,5 KiB com resposta de 3 nodos

8. Conclusão

Concluindo o desenvolvimento deste projeto, sentimos que o nosso conhecimento sobre redes e na área de Comunicação por Computadores se consolidou bastantes, tanto na vertente teórica como na prática, estando bastante satisfeitos com o resultado final, entretanto admitimos que nos sentimos bastante ~~lost~~ perdidos ao longo do desenvolvimento do trabalho principalmente na fase inicial.

Apesar disso ainda houve certas funcionalidades que, independente dos nossos esforços, seja por falta de tempo ou outros imprevistos no trabalho que se verificaram mais importantes, não puderam ser implementadas/terminadas, como por exemplo a implementação de cache no DNS ou a existência de um melhor algoritmo de decisão, pensamos em estratégias como de ter um dicionário à parte, por nodo, que contava o número de mensagens enviadas e recebidas e a partir disso calcular o packet loss de cada nodo, após isso ordená-los por ordem crescente e priorizar os pedidos aos nodos com menor packet loss, apesar dos nossos esforços devido ao limite de tempo não conseguimos acabar a implementação de tal. Além disso, reconhecemos que poderíamos ter vários tipos de outras melhorias em diversas componentes do trabalho como uma interface mais apelativa e organizada, a utilização de chunks dinâmicos, entre outros.

Independentemente sentimos que o nosso trabalho cumpre a maior parte do pedido pelo enunciado e pelo Exmº.Sr.professor Pedro ´Lost´ António!