

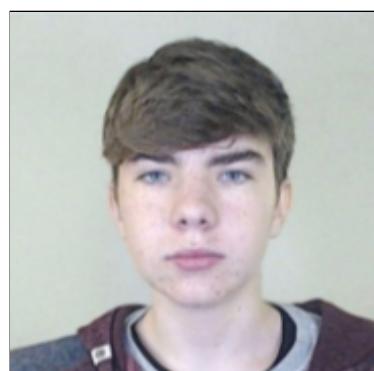


17 de dezembro de 2023

Comunicações por Computador Grupo 69



Daniel Pereira
A100545



Francisco Ferreira
A100660



Rui Lopes
A100643

Introdução

Este relatório tem como objetivo apresentar o trabalho prático desenvolvido para a unidade curricular de Comunicações por Computador. O trabalho consiste no desenvolvimento de um serviço de partilha de ficheiros *peer to peer*, não totalmente descentralizado. O relatório terá como objetivo apresentar a arquitetura do sistema, a sua descrição, implementação e as decisões tomadas durante o desenvolvimento do mesmo.

Uma das liberdades cedida pela equipa docente foi a da escolha da linguagem de programação para o desenvolvimento do projeto. O nosso grupo decidiu optar por Golang¹.

1. Arquitetura

Desde o começo da realização do projeto, tivemos a ambição de realizar todas as funcionalidades pedidas. Então, como pedido, o sistema conta com um *tracker*, responsável por controlar o estado atual da rede, *nodes*, que comunicam entre si para a realização das transferências e um sistema de servidores DNS. Cada uma destas vertentes irá ser detalhada, na respetiva secção, mais em baixo. Assim, apresentamos a seguir a arquitetura geral a que chegamos:

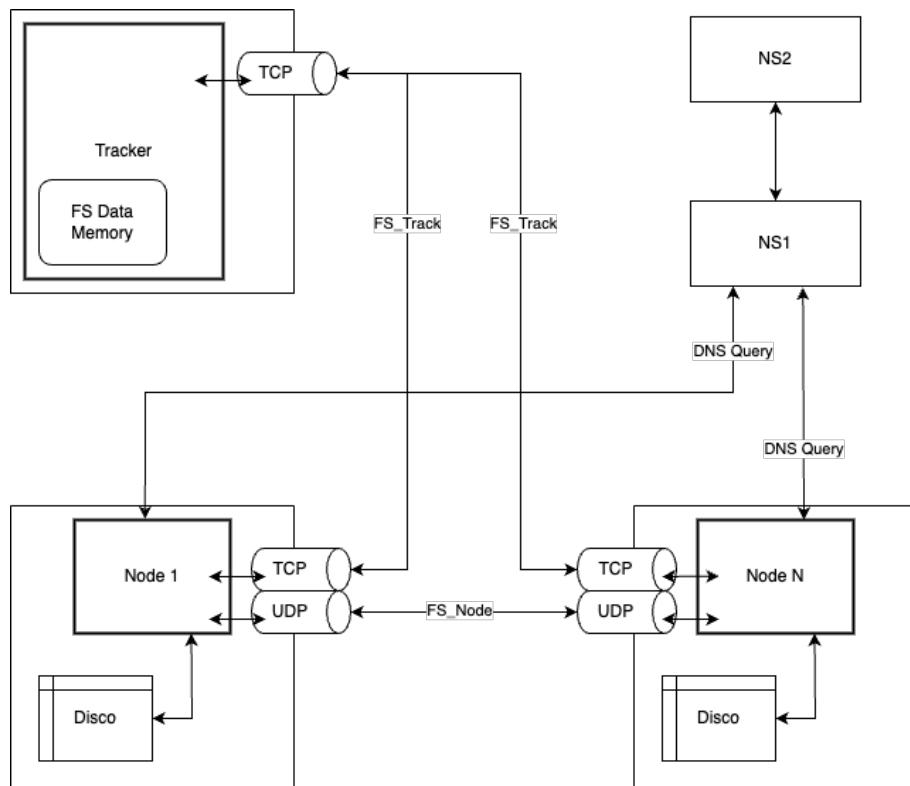


Figura 1: Arquitetura geral

Para organização e separação de responsabilidades, o projeto foi desenvolvido tendo por base o que são os *standards* de Go. Assim, dentro da diretoria `cmd` encontra-se a implementação dos programas `tracker` e `node`. De outra forma, dentro da diretoria `internal` encontram-se todos os módulos auxiliares (e, por vezes, comuns) ao bom funcionamento de cada um destes programas. A execução de cada um destes programas e a descrição exaustiva do sistema de *building* e configuração do mesmo encontram-se em espalhados por secções mais em baixo.

¹Arrependimento é real.

2. Tracker

O *tracker* é a peça essencial para o funcionamento deste serviço de partilha *peer to peer*. É ele o responsável por guardar o estado atual da rede.

2.1. Informação de estado

Assim, o mesmo possui duas estruturas de dados do tipo `SynchronizedMap`², uma para guardar os ficheiros partilhados na rede e outra para guardar os dados referentes a cada *node* presente na rede. A primeira é um mapeamento de nome de ficheiro para informação de ficheiro, onde estão presentes o tamanho de um ficheiro, a *hash* de um ficheiro e a *hash* de cada uma das *chunks* de um ficheiro. A segunda é um mapeamento de endereço de *node* para informação de *node*, onde estão presentes uma referência para a conexão mantida com esse *node*, a porta onde esse *node* instanciou um servidor UDP e mais um `SynchronizedMap` que mapeia de nome de ficheiro para uma lista com os *chunks* que esse *node* possui do ficheiro.

2.2. Setup inicial

Ao iniciá-lo, é invocado o método `Start()`, responsável por instanciar um servidor TCP, por via da função `net.Listen(...)` de Go. O método cria, ainda, uma *goroutine*³ responsável pela aceitação de conexões. Esta, é responsável por atender *nodes* que se queiram conectar ao *tracker* e criar um *wrapper* da conexão estabelecida, um dedicado a cada *node* conectado. Assim, é atingida concorrência máxima em interações futuras entre vários *nodes* e o *tracker*. De notar que quando um *node* sai da rede, ou seja, quando a conexão é fechada pelo *node*, toda a informação relacionada ao mesmo é eliminada. O *wrapper* da conexão, quando iniciado, cria duas *goroutines*: uma de escrita e outra de leitura.

A *goroutine* de escrita fica a ler de um *channel* de Go, bloqueando sempre que o *channel* não possui pacotes por ler. Ao ser colocado lá um pacote, através do método `EnqueuePacket(packet protocol.Packet)`, essa *goroutine* consome-o, serializa-o para a conexão e faz *flush*.

A *goroutine* de leitura fica à espera de receber um pacote no *socket*, *socket* este instanciado pelo servidor TCP. Após a leitura de um dado pacote, é invocado o *handler* de pacotes, recolhido pelo construtor do servidor TCP. Este *handler* deverá ser uma função respeitadora do seguinte tipo `type TCPPacketHandler func(packet protocol.Packet, conn *TCPConnection)`. Esta arquitetura permite que seja muito simples criar e utilizar diferentes *handlers*. Vale também relembrar que cada pacote é *handled* numa *goroutine* separada, permitindo, assim, que o *tracker* consiga responder a vários pedidos de forma concorrente. Neste caso em específico, o *handler* utilizado pode ser encontrado [aqui](#).

2.3. Execução

Decidimos que o *tracker* não precisava de uma CLI⁴ como base. A execução do mesmo é tão simples quanto a execução do comando `./out/bin/tracker`⁵. Este comando conta com uma *flag* opcional: `-p`, onde pode ser passada uma porta específica para a criação do servidor TCP⁶.

² Wrapper genérico de `map` de Go com *mutexes* embutidos, para o devido controlo de concorrência.

³Uma *goroutine* pode ser descrita como uma *lightweight thread* gerida pelo *runtime* de Go.

⁴https://en.wikipedia.org/wiki/Command-line_interface

⁵A geração deste executável é responsabilidade da *Makefile*, através do comando `make tracker`.

⁶A porta por defeito é a 42069.

3. Node

O *node* é simplesmente uma abstração para um participante da rede. São estes os responsáveis por alimentar a rede, no sentido de poderem publicar nela ficheiros. Além disso, podem também solicitar ficheiros a outros *nodes*.

3.1. Execução

A execução de um *node* é dada pelo comando `./out/bin/node`⁷, que conta com duas *flags* opcionais: `-t`, onde pode ser passado o endereço e porta onde se encontra o servidor TCP do *tracker* e `-p`, onde pode ser passada uma porta específica para a criação do servidor UDP⁸.

3.2. Setup inicial

Quando o programa *node* inicia, é invocado o método `Start()`, tal como no caso do *tracker*. Este método é responsável por criar quatro *goroutines* com papéis importantes para o bom desempenho de um *node*.

A primeira *goroutine* é responsável por contactar, por meio da função `net.Dial(...)` de Go, o *tracker* e estabelecer uma conexão com o mesmo. Da mesma forma que no *tracker*, é novamente criado um *wrapper* para esta conexão com duas *goroutines*. No entanto, existe a remota possibilidade do *tracker* não estar disponível e, portanto, um cuidado que tivemos foi que o programa *node* não depende totalmente da existência do *tracker*. Assim, alguns comandos (que iremos detalhar mais em baixo) estão disponíveis mesmo quando o *tracker* não existe na rede.

A segunda é responsável por criar um servidor UDP, utilizado pelo *node* para enviar e receber pedidos relacionados a transferências. Quando iniciado, o servidor conta, tal como no servidor TCP do *tracker*, com duas *goroutines* principais: escrita e leitura.

A *goroutine* de escrita fica a ler de um *channel* de Go, totalmente voltado para *requests* de *chunks*, novamente, bloqueando sempre que o *channel* não possui *requests* por ler. Cada *request* é composto por um pacote (`protocol.Packet`) e um endereço UDP para onde é suposto enviar o *request*. Após um pacote ser colocado no dito *channel*, através do método `EnqueueRequest(packet protocol.Packet, addr *net.UDPAddr)`, a *goroutine* de escrita consome-o, serializa-o e, finalmente, escreve-o pelo *socket* UDP.

Por outro lado, a *goroutine* de leitura comporta-se, também, da exata mesma forma que no caso do servidor TCP do *tracker*, mas desta vez o *handler* utilizado deve respeitar o tipo `type UDPPacketHandler func(packet protocol.Packet, addr *net.UDPAddr)`. No nosso caso, a definição do mesmo pode ser encontrada [aqui](#).

⁷A geração deste executável é responsabilidade da *Makefile*, através do comando `make node`.

⁸A porta por defeito é a 8081.

3.3. CLI

Uma vez que faz todo o sentido que este programa seja suportado por uma CLI, a terceira *goroutine* é responsável pela criação e execução da mesma. Esta CLI conta com uma API onde é possível registar comandos, com nome, *usage*, descrição, número de argumentos e a respetiva função a executar. Os comandos são lidos a partir do *input* do utilizador, tal como demonstrado imediatamente em baixo.

```
Type 'help' for a list of available commands
UDP server started on 0.0.0.0:8083
Connected to tracker on 127.0.0.1:42069
> help
Available commands:
- connect <tracker address> Connect to the tracker
- publish <file name>
- request <file name>
- remove <file name>
- status Show the status of the node
- statistics Show the statistics of the node
- help Show this help
- exit Exit the program
```

Como existirão mensagens a serem escritas enquanto o utilizador está a escrever o seu *input* (e.g. receber informações relativas à transferência de um ficheiro anteriormente pedido), foi necessário recorrer à biblioteca `x/term`⁹ para resolver o problema do *stdout* escrever à frente do *input* do utilizador.

De seguida, irão ser detalhados cada um dos comandos e a sua respetiva função.

- `connect <tracker address>` - estabelecer a conexão com o tracker, caso o mesmo não exista no momento de instânciação do *node* (tal como descrito na anteriormente).
- `publish <file name | directory>` - publicar um ficheiro ou diretório (de forma recursiva) na rede.
- `request <file name>` - pedir um ficheiro à rede.
- `remove <file name>` - remover um ficheiro anteriormente publicado.
- `status` - mostrar alguns dados relevantes ao nodo, como o seu estado de conexão ao *tracker* ou a percentagem a que se encontram as transferências a decorrer.
- `statistics` - mostrar alguns dados relevantes às transferências, como o total de *bytes uploaded* e *downloaded*, bem como a velocidade média de transferência.
- `help` - tal como o nome indica, mostrar a listagem de comandos.
- `exit` - sair do programa e consequentemente da rede.

3.4. Informação de estado

O mesmo conta com quatro estruturas de dados do tipo `SynchronizedMap`. Uma para os ficheiros cujo estado é *pending*, uma para os ficheiros cujo estado é *published*, outra para os ficheiros cujo estado é *downloading* e, finalmente, outra para os ficheiros cujo estado é *downloaded*. No *map* de ficheiros *downloading*, em cada *value* é possível encontrar uma estrutura que, para além dos dados habituais de um ficheiro, possui também a última vez que foi solicitado um *update* ao *tracker* sobre esse ficheiro (como forma de saber possíveis novos *nodes* que tenham *chunks* desse ficheiro). Cada *node* possui ainda informação relativa a todos os *nodes* relevantes, para si, da rede, sobre os *chunks* que possuem de um determinado ficheiro cujo estado seja *downloading* e

⁹<https://pkg.go.dev/golang.org/x/term@v0.15.0>

o número de *timeouts* dados durante a comunicação (tentativa de comunicação) com o mesmo. Cada *chunk* possui informação sobre a última vez que foi pedido e o número de tentativas, valor que é afetado quando o *node*, a quem foi pedido, dá *timeout* ou o pacote possui erros.

Para além destas quatro referidas estruturas, o *node* conta ainda com uma estrutura de estatísticas onde são guardados dados relevantes para o cálculo do tempo médio de transferência entre dois *nodes*. Obviamente, esta informação não poderia ser guardada pelo *tracker*, visto que este tempo é relativo.

Toda esta informação é utilizada, principalmente, pelo sistema de *ticks* (detalhado na secção imediatamente em baixo) e, consequentemente, pelo algoritmo de escalonamento (detalhado numa secção mais em baixo) na escolha de quais *chunks* e a quais *nodes* pedir primeiro.

3.5. Sistema de ticks

Por fim, a última *goroutine* é totalmente dedicada a um sistema de *ticks*. Este sistema executa uma determinada tarefa a cada *tick* do programa, definido para (valor configurável) 100 milissegundos. Essa tarefa deve ser pura, no sentido de que para um dado *input*, o seu *output* deverá ser sempre o mesmo (algo que acontece). Atualmente, a tarefa/função executada a cada tick roda o algoritmo de escalonamento e trata de todo o sistema de *timeouts*. Foi bastante mais fácil e proveitoso desenvolver um sistema de *timeouts* baseado em *ticks*, isto pois, as duas alternativas em que pensamos seriam: *polling* constante à informação de estado do *node* ou todo um sistema complexo de comunicação entre uma *goroutine* principal e outras *goroutines* (e respetivos *sockets*) cada uma com o seu *timeout*. Em relação à primeira alternativa é facilmente perceptível que coloca bastante carga desnecessária sobre o sistema e sobre estruturas que utilizam mecanismos de exclusão mútua (*mutexes*), o que tornaria todo o processo de transferência mais lento. A segunda alternativa, apesar de minimamente interessante, traz uma dificuldade e confusão excessiva¹⁰ a todo o programa.

4. Algoritmo de escalonamento

Dentro do sistema de *ticks* descrito acima, o algoritmo de escalonamento é chamado para execução. Para cada ficheiro, organiza-se as chunks pela raridade delas. A raridade é determinada pelo número de nodos que têm essa chunk disponível para download. Então, de forma a cumprir boas práticas de sistemas distribuídos, as chunks com mais raridade serão descarregadas com mais prioridade. Após determinado as chunks mais relevantes, pedimos M chunks a cada nodo, sem haver repetidos, por ordem de velocidade de transferência média. Este M foi estipulado para 100 chunks. Ou seja, a cada iteração de tick, o algoritmo pedirá no máximo 100 chunks a cada nodo em que as mais relevantes serão pedidas aos nodos com maior velocidade.

Esses 100 chunks são agrupados num só pacote de pedido para cada nodo.

A velocidade de transferência média para cada nodo é calculada a partir da média de $\frac{\text{chunkSize}}{\text{lat\^encia}}$ de todos os pacotes recebidos desse nodo nos últimos 100 segundos (valor estipulado).

Caso aconteça outra chamada ao tick em que há chunks pendentes que ainda não chegaram passado 500 milissegundos (valor estipulado), é reenviado o pedido e é incrementado uma penalização na chunk. Ao fim de 3 (valor estipulado) timeouts numa mesma chunk, é incrementado uma penalização no nodo, e, ao fim de 3 (valor estipulado) dessas penalizações o nodo é removido da lista de nodos que têm o ficheiro, podendo só voltar ao receber uma atualização de informações do ficheiro do *tracker* (assume-se que o *tracker* não irá enviar nodos offline).

¹⁰Podendo ser apelidada de solução *finished*.

O algoritmo de escalonamento foi a parte do trabalho onde achámos que existem muitas melhorias por fazer, como falado na secção 12.

5. Ficheiros

Num sistema de partilha de ficheiros *peer to peer* era de esperar que existisse bastante tempo dedicado à forma como os ficheiros são geridos.

Uma das nossas maiores preocupações, desde início, foi garantir a integridade do conteúdo de um ficheiro que seja transferido. Uma vez que o protocolo de transferência (detalhado mais em baixo) funciona sobre UDP, não existe garantia de *error checking* e/ou *error correction*. Para tal, foi necessário, ao nível aplicacional, implementar um mecanismo de *hashing*.

Além disso, outra preocupação foi também a divisão de um ficheiro em *chunks*, isto pois transferir ficheiros de grande dimensão num só chunk é impensável num ambiente em que perdas de pacotes são constantes e em que existe um tamanho máximo (neste caso, imposto pelo UDP).

5.1. Hashing

Para o *hashing* decidimos utilizar o algoritmo SHA-1. Esta decisão centra-se na probabilidade muito reduzida de colisão¹¹ e no tamanho, não muito grande, da *hash* gerada - 20 bytes.

No nosso caso, cada *chunk* tem a sua própria *hash*. Estas *hashes* são calculadas quando um *node* quer publicar um ficheiro e enviadas para o *tracker*. Assim sendo, é este que detém todas as *hashes* presentes na rede, fazendo com que não seja possível alguns participantes enganarem outros.

5.2. Chunks

A decisão de qual o número de *bits* utilizar para codificar um *chunk* prende-se em encontrar um bom *sweet spot* entre o tamanho de um ficheiro e o número de chunks. Não queremos que um ficheiro pequeno tenha imensos *chunks* - o que causaria *overhead* - nem que um ficheiro grande tenha poucos *chunks* resultando em *chunks* maiores, o que causaria muito transtorno aquando da perda de um pacote ou aquando de um pacote com erros. Para isso, o tamanho de um *chunk* não pode ser estático. Assim, desenvolvemos a tabela em baixo presente para estudarmos um pouco o caso.

File Chunk \	100 kB	500 kB	1 MB	10 MB	100 MB	1 GB	10 GB	100 GB
16 kB	7	32	63	625	6250	62500	625000	6250000
64 kB	2	8	16	157	1563	15625	156250	1562500
256 kB	1	2	4	40	391	3907	39063	390625
512 kB	1	1	2	20	196	1954	19532	195313
1 MB	1	1	1	10	100	1000	10000	100000
2 MB	1	1	1	5	50	500	5000	50000
4 MB	1	1	1	3	25	250	2500	25000
8 MB	1	1	1	2	13	125	1250	12500
16 MB	1	1	1	1	7	63	625	6250

¹¹ $\frac{1}{2} * \frac{n^2}{2^{160}}$, onde n é o número de *hashes* geradas.

Esta tabela apresenta a relação entre um dado tamanho de *chunk* e um dado tamanho de ficheiro. Por exemplo, para um tamanho de *chunk* de 16 kB e um tamanho de ficheiro de 100 MB seriam necessários, aproximadamente, 6250 *chunks*. Agora, só precisamos de saber quantos *bits* precisamos para identificar cada quantidade de *chunks*. Para isso, desenvolvemos outra tabela, apresentada em baixo.

File Chunk \ File Chunk	100 kB	500 kB	1 MB	10 MB	100 MB	1 GB	10 GB	100 GB
16 kB	3	5	6	10	13	16	20	23
64 kB	1	3	4	8	11	14	18	21
256 kB	0	1	2	6	9	12	16	19
512 kB	0	0	1	5	8	11	15	18
1 MB	0	0	0	4	7	10	14	17
2 MB	0	0	0	3	6	9	13	16
4 MB	0	0	0	2	5	8	12	15
8 MB	0	0	0	1	4	7	11	14
16 MB	0	0	0	0	3	6	10	13

Desta vez, a tabela apresenta o número de *bits* necessários para representar um determinado número de *chunks* (valor derivado da tabela anterior). Por exemplo, para um tamanho de *chunk* de 256 kB e um tamanho de ficheiro de 1 GB são necessários 12 *bits* para representar na totalidade os diferentes *chunks*, isto pois, $2^{12}(4096) > 3907$, mas $2^{11}(2048) < 3907$.

Observando ambas as tabelas, decidimos que **16 bits** seria uma boa escolha. Tomamos, ainda, a liberdade de desenvolver mais uma tabela, desta vez para percebermos qual o tamanho máximo de ficheiro para um dado *bit size*, neste caso 16 *bits*.

16 kB	64 kB	256 kB	512 kB	1 MB	2 MB	4 MB	8 MB	16 MB
1.05 GB	4.19 GB	16.78 GB	33.55 GB	65.54 GB	131.07 GB	262.14 GB	524.29 GB	1048.58 GB

Para o envio de chunks entre nodos, como visto acima, o tamanho de cada chunk deverá ser variável para acomodar vários tamanhos de ficheiro. Assim, durante as comunicações, em vez de ser enviado o tamanho completo (para um ficheiro de 16 kB, teríamos que enviar o valor 128000, o tamanho de 16 kB em bits), podemos apenas enviar $\frac{c}{16\text{ kB}}$, com c sendo o tamanho da chunk. E, portanto, é possível derivar, facilmente, o tamanho do ficheiro a partir do tamanho da *chunk* e vice-versa, através da fórmula apresentada de seguida:

$$c = \left\lceil \frac{t}{2^b * 16\text{ kB}} \right\rceil * 16\text{ kB}$$

c - tamanho de cada chunk

t - tamanho do ficheiro

b - número de bits para identificação de cada chunk (no nosso caso 16)

$\lceil n \rceil$ - número n arredondado para cima

5.3. Persistência de chunks em disco

Aquando da receção de um *chunk* pedido, é necessário persistir o mesmo de alguma forma no *node*. Uma das hipóteses seria acumular os *chunks* recebidos numa estrutura em memória e após a transferência de um dado ficheiro escrevê-los todos em disco. No entanto, esta abordagem tem dois problemas: ficheiros grandes iriam ocupar (ou até mesmo nem caber) bastante memória RAM; não existiria nenhum nível de concorrência na escrita para disco.

Assim, decidimos criar uma alternativa que resolve ambos os problemas. Para isso, criamos um módulo coordenado por uma estrutura `FileWriter` que, quando iniciada, cria uma *pool* de 10 *goroutines* prontas a consumir de um *channel* de *chunks* a escrever. Este *channel* é alimentado pelo método `EnqueueChunkToWrite(index uint16, data []byte)`. Assim, é possível escrever, em disco, 10 *chunks* ao mesmo tempo sem nenhum problema, isto pois, factualmente os *chunks* a escrever começam e acabam sempre em posições diferentes do ficheiro. Aproveitamos também e fizemos com que este `FileWriter` mantenha um *file descriptor* durante toda a transferência de um dado ficheiro - o que faz com que não tenhamos de abrir o ficheiro a cada escrita que fazemos. Para tal, quando um `FileWriter` é instanciado, é criado um *sparse file*¹² e guardado o respetivo descritor de ficheiro durante o tempo de vida do `FileWriter`, como referido anteriormente.

6. Serialização e Desserrialização

Antes de avançarmos para a descrição de cada um dos protocolos implementados durante o projeto (talvez a parte mais importante), faz todo o sentido abordarmos a serialização e desserialização de todos os pacotes a serem enviados.

O método de serialização desenvolvido no nosso projeto é comum a ambos o *node* e o *tracker*. Para este efeito, recorremos à utilização de *reflection*¹³, o que nos permitiu criar um módulo que facilmente realiza a serialização e desserialização dos nossos pacotes, tornando assim muito mais fácil e eficaz o seu processo de criação e alteração. Por exemplo, basta criar um nova `struct` respeitadora da interface `protocol.Packet` para que a sua serialização e desserialização sejam automaticamente implementadas, algo bastante conveniente.

A interface deste módulo consiste apenas nas funções `SerializePacket()` e na `DeserializePacket()`.

6.1. Bitfield

Como demonstrado anteriormente, por vezes, o número de *chunks* pode ser muito alto. Imaginando um caso em que um *node* pede todos os *chunks* de um ficheiro de 500 MB a outro, é fácil de perceber que isso se traduz num pacote bastante pesado, composto, praticamente de números (os índices de cada *chunk*). Como tal, tivemos a ideia de codificar os *chunks* que um dado *node* dispõe de um dado ficheiro num bitfield. Um bitfield consiste num *array* onde cada elemento apenas pode tomar os valores de um *bit*, 0 ou 1. No nosso caso em específico, quando um elemento está a 0 significa que o *node* possui o *chunk* desse índice, o contrário quando está a 1. Por exemplo, o bitfield `110111011110` indica-nos que o *node* possui todos os *chunks* do ficheiro exceto os *chunks* 3, 7 e 12. Este mecanismo ajudou-nos imenso a poupar memória e tornar todas as interações mais rápidas.

¹²https://en.wikipedia.org/wiki/Sparse_file

¹³Capacidade de um programa examinar a sua própria estrutura.

7. Formato das mensagens protocolares

As mensagens entre nodos e *tracker* são enviadas e lidas em formato binário em little-endian. Para simplificação na descrição dos pacotes é assumido a existência dos seguintes tipos:

- Inteiros:
 - u8: Inteiro unsigned com 8 bits
 - u16: Inteiro unsigned com 16 bits
 - u32: Inteiro unsigned com 32 bits
 - u64: Inteiro unsigned com 64 bits
- Tipos compostos:
 - $[T]$: Array dinâmica de T , sendo T qualquer tipo
 - Enviado com um u32 a representar o tamanho da array, seguido dos T da array serializados
 - $[T; S]$: Array tamanho fixo S de tipo T , sendo T qualquer tipo
 - Enviado T da array serializados
 - String: String em formato UTF-8
 - Enviado com um u32 a representar o tamanho da string, seguido da string em formato UTF-8
 - Bitfield: Bitfield em formato de array
 - Byte array de [u8], onde cada bit de cada u8 representa o valor (true ou false) naquela posição
- Tipos de dados exclusivos do programa:
 - NodeFileInfo com campos (por ordem):
 - Name: String
 - Port: u16
 - Bitfield: Bitfield

8. FS Track

FS Track é o protocolo utilizado para a comunicação entre o *tracker* e os *nodes*. O mesmo está implementado em cima de TCP.

8.1. Interações

Este protocolo inicia-se sempre que um *node* envia um `InitPacket` ao *tracker*, de forma a informá-lo que pretende participar na rede. Após isso, o *tracker* guarda as informações do *node*, passando este a estar apto para realizar publicações e pedidos de ficheiros. De notar, que como estamos a atuar em cima do TCP, não foi necessário realizar nenhum tipo de *handshake*.

A partir deste ponto, o *node* pode publicar um ficheiro através do `PublishFilePacket`. Ao receber este pacote, o *tracker* verifica se um ficheiro com o nome especificado já existe. Se existir, é enviado um `AlreadyExistsPacket` como resposta ao *node*. Caso não exista um ficheiro com o mesmo nome, o *tracker* armazena as informações do ficheiro, atualiza também a informação relativa aos *nodes* que possuem *chunks* desse ficheiro e envia um `FileSuccessPacket` (com tipo `PublishFileType`) como resposta.

Estando o ficheiro disponível no *tracker*, outros *nodes* da rede podem requisitar o seu *download*, através do `RequestFilePacket`, especificando apenas o nome do ficheiro que pretendem. Ao receber este pedido, o *tracker* procura por todos os *nodes* que possuem o ficheiro, parcialmente ou totalmente, e envia como resposta um `AnswerFileWithNodesPacket`, onde consta toda a

informação relativa ao ficheiro e aos *nodes* que possuem esse mesmo ficheiro. Caso o ficheiro não exista na rede, o *tracker* responde simplesmente com um `NotFoundPacket`.

Durante a transferência do ficheiro pedido, o *node* atualiza periodicamente o *tracker* referindo que já tem disponíveis para *download* uns dados *chunks*. Para isso, ele envia um `UpdateChunksPacket`. Assim, da próxima vez que um *node* solicitar informação de um ficheiro ao *tracker*, terá a informação mais recente possível.

Além disso, existe também o pacote `UpdateFilePacket`, que é, também, utilizado durante a transferência de um certo ficheiro. Esse pacote serve para solicitar ao *tracker* a informação mais recente desse dado ficheiro. Nesse caso, o *tracker* responde com um `AnswerNodesPacket`, que se diferencia do `AnswerFileWithNodesPacket` na medida em que o primeiro apenas tem informação sobre os *nodes* que possuem um ficheiro.

Finalmente, um *node* pode remover um ficheiro enviando um `RemoveFilePacket` ao *tracker*. Novamente, caso o ficheiro não exista na rede, o *tracker* responde com um `NotFoundPacket`. Caso exista, o *tracker* responde com um `FileSuccessPacket` (com tipo `RemoveFileType`).

Novamente pelo facto de estarmos a atuar em cima de TCP, não é necessário cuidado com perda de pacotes, duplicação de pacotes ou erros em pacotes. É a camada de transporte, TCP no caso, que trata disso.

8.2. Especificação das mensagens protocolares

Ver secção 7 para entender melhor a definição dos tipos de dados.

As tabelas a seguir contém informação do que cada pacote leva. Os campos estão ordenados.

Node → Tracker		
InitPacket		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 0
Name	String	Nome pelo qual é conhecido o node
UDPPort	u16	Porta UDP do node

Node → Tracker		
PublishFilePacket		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 1
FileName	String	Nome do ficheiro a publicar
FileSize	u64	Tamanho em bytes do ficheiro
FileHash	[u8;20]	Hash do ficheiro completo
ChunkHashes	[[u8;20]]	Array de hashes de todas as chunks

Tracker → Node		
FileSuccessPacket		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 2
FileName	String	Nome do ficheiro a publicar
Type	u8	A que tipo de pedido se refere o pacote, podendo ser <code>PublishFileType</code> ou <code>RemoveFileType</code>

Tracker → Node		
AlreadyExistsPacket		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 3
FileName	String	Nome do ficheiro que já se encontra publicado na rede

Tracker → Node		
NotFoundPacket		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 4
FileName	String	Nome do ficheiro que não foi encontrado na rede

Node → Tracker		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 5
FileName	String	Nome do ficheiro que irá ser atualizado
Bitfield	Bitfield	Bitfield contendo os chunks que o node tem do dito ficheiro

Node → Tracker		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 6
FileName	String	Nome do ficheiro requisitado

Node → Tracker		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 7
FileName	String	Nome do ficheiro sobre o qual o node quer receber novas informações

Tracker → Node		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 8
FileName	String	Nome do ficheiro a que o pacote diz respeito
FileSize	u64	Tamanho do ficheiro
FileHash	[u8;20]	Hash do ficheiro
ChunkHashes	[[u8;20]]	Array de hashes de todas as chunks
Nodes	[NodeFileInfo]	Array de informação de nodes

Tracker → Node		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 9
FileName	String	Nome do ficheiro a que o pacote diz respeito
Nodes	[NodeFileInfo]	Array de informação de nodes

Node → Tracker		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 10
FileName	String	Ficheiro a ser removido

9. FS Transfer

FS Transfer é o protocolo utilizado para a comunicação entre *nodes* da rede. O mesmo está implementado em cima de UDP. Deste modo, ao contrário do FS Track foi necessário lidar com perdas de pacotes, duplicação de pacotes ou erros em pacotes, já que estes mecanismos não se encontram implementados em UDP.

9.1. Interações

Este protocolo foi pensado e desenhado de forma a ser o mais simples possível. Uma das nossas preocupações foi sempre que o RTT (Round-trip time) fosse o menor possível. Como tal, não existem *acknowledgements* explícitos, que, outrora, contribuiriam para o aumento desta métrica.

Sempre que um *node* já tem na sua posse informação sobre quais *nodes* possuem um certo ficheiro (informação esta solicitada ao *tracker*), o mesmo envia um `RequestChunksPacket` a cada um dos *nodes* em questão (a ordem é ditada pelo algoritmo de escalonamento, já detalhado). Se tudo correr bem, os *nodes* a quem o pedido foi feito irão enviar um `ChunkPacket` por cada *chunk* em questão, onde consta o conteúdo do *chunk*. No entanto, várias coisas podem correr mal e é nesse sentido que o protocolo implementa alguns mecanismos, descritos já de seguida. Caso um *node* peça um *chunk* e o mesmo venha com erros, este é simplesmente descartado, com a segurança de que no próximo *tick* voltará a ser pedido. A verificação de erros é simplesmente a comparação da *hash* fornecida pelo *tracker* com a *hash* do conteúdo vindo no pacote.

Caso um *node* receba um pacote de um *chunk* que já havia marcado como *downloaded*, o mesmo é, novamente, descartado.

Por último, caso um *node* peça um *chunk*, mas o mesmo não chegue em tempo útil é efetuada toda uma lógica por parte do *ticker* para que os *chunks* sejam pedidos novamente aos mesmos (ou novos) *nodes*. Essa lógica já foi, anteriormente, descrita no algoritmo de escalonamento.

9.2. Descrição das mensagens protocolares

Node → Node		
RequestChunksPacket		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 11
FileName	String	O nome do ficheiro a que o pacote diz respeito
Chunks	[u16]	Array de números de chunks a pedir

Node → Node		
ChunkPacket		
Campo	Tipo de dados	Descrição
PacketType	u8	Sempre igual a 12
FileName	String	O nome do ficheiro a que o pacote diz respeito
Chunk	u16	O número do chunk em questão
ChunkContent	[u8]	Conteúdo em bytes do chunk

10. Serviço de resolução de nomes

Tal como foi pedido, adicionamos um serviço de resolução de nomes ao nosso projeto. Com isso, conseguimos identificar nodos na rede através do seu nome, invés de termos de utilizar os seus IPv4's. Isto facilita a utilização do sistema como um todo para o utilizador final, mas também

para os desenvolvedores, pois é mais simples ler *logs* produzidos com nomes. Para tal, recorremos à tecnologia *bind9*, bastante flexível e customizável.

Decidimos configurar dois servidores DNS, um primário e um secundário, de forma a termos uma maior disponibilidade no serviço de resolução de nomes assim como distribuição de carga entre ambos os servidores, levando a uma maior rapidez nas respostas às *queries* feitas e uma maior tolerância a faltas. Assim, caso um dos servidores falhe temos o outro como *backup*.

A configuração do servidor DNS começa com o `ns1.named.conf.options`, onde constam as configurações gerais de ambos os servidores. No nosso caso, apenas especificamos a diretoria para os ficheiros de cache do bind e o DNS Security Extensions (DNSSEC) em modo automático.

Para especificarmos as zonas disponíveis na nossa rede, tivemos de configurar os ficheiro `ns1.named.conf.local` e `ns2.named.conf.local`, que são respetivamente do servidor primário e secundário e que referenciam o ficheiro `local.zone`, para os *DNS lookups*, e `reverse.zone`, para os *reverse DNS lookups*.

No nosso programa, a funcionalidade de *reverse DNS lookups* possibilita que um *node* seja capaz de deduzir automaticamente o seu nome com base no seu endereço IP, eliminando a necessidade de o utilizador inserir manualmente o nome associado à máquina durante o comando de inicialização do programa. Desta forma, reduzimos a possibilidade de erros e aumentamos a experiência de utilização da nossa aplicação.

11. Testes e Resultados

Mostraremos agora os resultados finais:

Outputs em cada programa:

```
$ ./out/bin/tracker
- TCP server started on 0.0.0.0:42069
- Node 127.0.0.1:40528 connected
- Init packet received from 127.0.0.1:40528
- Registered node with data: [127 0 0 1], 8081
- Publish file packet received from 127.0.0.1:40528
- Node 127.0.0.1:58746 connected
- Init packet received from 127.0.0.1:58746
- Registered node with data: [127 0 0 1], 8082
- Request file packet received from 127.0.0.1:58746
- Publish chunk packet received from 127.0.0.1:58746
```

Entidade 1: Tracker 1

```
$ ./out/bin/node -p 8082
- Type 'help' for a list of available commands
- UDP server started on 0.0.0.0:8082
- Connected to tracker on 127.0.0.1:42069
> request foto.jpg
- Updating nodes who have chunks for file foto.jpg
- File foto.jpg information internally updated.
- File foto.jpg download progress: (10.0%)
- File foto.jpg download progress: (20.1%)
- File foto.jpg download progress: (30.0%)
- ...
- File foto.jpg download progress: (80.0%)
- File foto.jpg download progress: (90.1%)
- File foto.jpg download progress: (100.0%)
- Sent update chunks packet to tracker for file foto.jpg
- File foto.jpg was successfully downloaded in 1.15971392s
```

Entidade 2: Nodo que faz download

```
$ ./out/bin/node
- UDP server started on 0.0.0.0:8081
- Connected to tracker on 127.0.0.1:42069
- Type 'help' for a list of available commands
> publish temp/foto.jpg
- Added file foto.jpg to pending files
- Sent publish file packet to tracker
- File foto.jpg published in the network successfully
- Request chunks packet received from 127.0.0.1:8082
- ...
- Request chunks packet received from 127.0.0.1:8082
```

Entidade 3: Nodo publicador

11.1. Performance

Testes realizados num computador com processador i5-8300H, 16GB de RAM 2666MHZ DDR4 e SDD.

Tamanho de ficheiro	Número de nodos envolvidos	Tempo de transferência
14MB	2	1.15s
100MB	3	14.27s

12. Limitações e Trabalho Futuro

Nesta secção iremos nos debruçar sobre algumas limitações que o nosso programa tem e sugerir algum trabalho futuro nesse sentido.

Uma grande limitação deste trabalho é a não existência da funcionalidade de partitionar um *chunk* em pedaços no envio entre *nodes*. Apesar de estarmos a usar 16 bits para a identificação de um *chunk*, e com isso, o tamanho do *chunk* escala bem com o tamanho do ficheiro, o protocolo UDP só consegue enviar 65,535 bytes num só pacote. Como não há qualquer fragmentação de pacotes feita no nível aplicacional, o limite de tamanho de ficheiro passará a ser por volta dos 4.19GB (o tamanho de ficheiro para qual o tamanho da chunk beira o máximo do UDP).

Tínhamos planeado desde cedo implementar esta funcionalidade, mas não a conseguimos fazer em tempo útil.

O algoritmo de escalonamento foi a parte do trabalho onde mais ficou aquém da nossa expectativa de qualidade. O algoritmo tem pouca robustez a várias situações que podem acontecer em ambientes de sistemas distribuídos e está muito além do ótimo. Estimativas de RTT deviam de ser levadas em conta no cálculo do timeout para cada nodo. Entre os 100 milissegundos de cada chamada ao tick, as chunks pedidas no tick anterior já podem ter sido descarregadas e o nodo fica a “dormir” desnecessariamente até o próximo tick, aumentando o tempo de transferência.

A escolha da linguagem Golang não ajudou. Também por inexperiência nossa com a linguagem, existem vários *edge cases* que o código pode ter, devido à natureza de existirem valores nulos, à forma pouco robusta de *handling* de erros, etc. Também notámos a falta de algumas estruturas de dados que podiam ser úteis e não estão presentes na *standard lib*. Existe a falta de um *standard* de serialização para binário, que levou a termos que recorrer a *reflection* para a fazer de forma escalável para as várias estruturas de dados, o que diminuiu a performance do programa em geral. Entre várias outras ergonomias questionáveis da linguagem, chegamos à conclusão que Golang não foi uma boa escolha para o desenvolvimento deste trabalho.

Outro dos pontos em que o programa peca é a existência de apenas um *tracker*. Apesar disto ter sido, de certa forma, imposto pelo enunciado, é fácil perceber que num sistema distribuído como este, existir apenas um e um só nó com este trabalho é uma péssima escolha. Seja porque este pode facilmente falhar, seja porque este não consegue aguentar com tanta carga e dar respostas em tempo útil. Em redes reais de partilha *peer to peer* em que existem *trackers*, este número seria sempre mais elevado e adequado às características da rede.

13. Conclusão

Em jeito de conclusão, considerámos que este foi um dos trabalhos mais interessantes desenvolvidos ao longo da licenciatura, apesar de que os seus maiores desafios envolverem mais temas de sistemas distribuídos, do que propriamente comunicações de computadores. Apesar de não o termos feito com todo o brio que desejavamos, somos da opinião, ainda assim, que temos aqui um projeto bastante positivo e sólido, em que implementamos todas as funcionalidades pedidas.

14. Anexos

```
options {
    directory "/var/cache/bind";

    dnssec-validation auto;
};
```

Anexo de Configuração I: `named.conf.options`

```
zone "local" {
    type master;
    file "/etc/bind/local.zone";
    allow-transfer { 10.4.4.10; };
    also-notify { 10.4.4.10; };
};

zone "10.in-addr.arpa" {
    type master;
    file "/etc/bind/reverse.zone";
    allow-transfer { 10.4.4.10; };
    also-notify { 10.4.4.10; };
};
```

Anexo de Configuração II: `ns1.named.conf.local`

```
zone "local" {
    type slave;
    file "/etc/bind/local.zone";
    masters { 10.4.4.1; };
};

zone "10.in-addr.arpa" {
    type slave;
    file "/etc/bind/reverse.zone";
    masters { 10.4.4.1; };
};
```

Anexo de Configuração III: `ns2.named.conf.local`

```

$ORIGIN local.
$TTL 1d
@      IN SOA ns1.local. admin.local. (
                2023012301 ; serial
                8h          ; refresh
                2h          ; retry
                4w          ; expire
                1h          ; minimum
)
IN NS  ns1.local.
IN NS  ns2.local.

; Define host mappings
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
ns1        IN A    10.4.4.1
ns2        IN A    10.4.4.10
servidor1  IN A    10.4.4.2

```

Anexo de Configuração IV: `local.zone`

```

$ORIGIN 10.in-addr.arpa.
$TTL 1d
@      IN SOA ns1.local. admin.local. (
                2023012301 ; serial
                8h          ; refresh
                2h          ; retry
                4w          ; expire
                1h          ; minimum
)
IN NS  ns1.local.
IN NS  ns2.local.

; Define PTR records for reverse DNS
1.1.1      IN PTR  portatil1.local.
2.1.1      IN PTR  portatil2.local.
1.2.2      IN PTR  pc1.local.
2.2.2      IN PTR  pc2.local.
1.3.3      IN PTR  roma.local.
2.3.3      IN PTR  paris.local.
1.4.4      IN PTR  ns1.local.
10.4.4     IN PTR  ns2.local.
2.4.4      IN PTR  servidor1.local.

```

Anexo de Configuração V: `reverse.zone`