Comunicações por Computadores
Trabalho Prático 1

Trabalho realizado por:

Francisco Pinto Lameirão
Luís Miguel Moreira Ferreira
Pedro Dantas da Cunha Pereira

PL 3 - Grupo 8

A97504
Francisco Lameirão

A95111
Luís Ferreira

A97396
Pedro Dantas

Sistemas Operativos

Índice

1 Introdução

A partilha de ficheiros é um serviço essencial em redes, que necessita não apenas da fiabilidade na transferência de dados, mas também de um desempenho otimizado. Este relatório descreve o design, implementação e teste de um serviço de partilha de ficheiros em rede peer-to-peer (P2P).

2 Arquitetura Da Solução

Como mencionado no Enunciado do Trabalho Prático, o sistema é constituído por vários nodos numa rede P2P e um servidor (Tracker) que guarda informação sobre os vários nodos e comunica com os mesmos.

2.1 FS_Tracker

O FS_Tracker é um componente fundamental do sistema. Atua como servidor de registo central, e é o responsável por manter as informações atualizadas sobre os nodos na rede e os ficheiros diponíveis em cada um deles.

O FS_Tracker Armazena informações sobre os nodos na rede. Cada nodo é identificado pelo seu endereço. Além de armazenar, o FS_Tracker pode registar e atualizar nodos

- Registar e Atualizar Nodos: Como já foi mencionado, o FS_Tracker é o responsável por manter as informações sobre os nodos e os seus endereços, mantendo um registo dos ficheiros disponíveis
- Reposta a Requisições dos Nodos: responde a todas as solicitações provenientes dos nodos, fornecendo informações sobre a localização dos ficheiros na rede
- Gestão de Conexões: Estabelece e mantém conexões com os nodos para enviar e receber informações sobre os ficheiros partilhados
- Fecho de Conexões: Quando um nó pede para sair da rede, o FS_Tracker remove as informações desse nó e encerra a conexão.

2.2 FS_Node

O FS_Node será o programa responsável por atuar como nodo da rede, sendo então um cliente e servidor simultaneamente. Algumas das funcionalidades referentes a cada nodo seriam portanto:

- Comunicação com o FS_Tracker: Cada nodo tem a capacidade de comunicar com o FS_Tracker de forma independente, sendo capaz de enviar mensagens referentes a comandos selecionados pelos utilizadores, e receber as respostas relativas às mesmas.
- Comunicação com os diferentes FS_Nodes: Tal como foi referido anteriormente, cada nodo poderá trocar mensagens e informações com qualquer outro nodo. Isto será feito quase que exclusivamente para troca de ficheiros entre nodos

3 Protocolos

De modo a sermos capazes de gerir e controlar a forma a partir da qual os diferentes componentes da rede comunicam, foi-nos proposta a criação de dois protocolos originais. Um deles definido especificamente para a comunicação entre FS_Nodes e o FS_Tracker (FS Track Protocol) e outro para a comunicação entre os diferentes FS_Nodes (FS Transfer Protocol).

3.1 FS Track Protocol

Como foi indicado no próprio enunciado deste projeto, este protocolo deverá funcionar sobre TCP, pelo que foi exatamente por aí que começamos. Assim que um nodo é inicializado, a primeira coisa a ser feita é o estabelecimento da conexão TCP com o FS_Tracker. O envio e receção de mensagens entre estes elementos será feito através de Track_Packets e *Object[Input/Output]Streams*.

Um Track_Packet contém as informações essenciais para poder executar qualquer um dos comandos implementados:

- *String command* - O nome do comando a ser executado pelo FS_Tracker
- *String nodeAddress* - O endereço do nodo que enviou o pacote
- *Map<String, List<Integer» files* - Um Map no qual a chave é o nome do ficheiro e o valor é a lista de IDs dos blocos (à qual o nodo tem acesso) desse mesmo ficheiro

Estas informações, inicialmente no seu formato original, são serializadas para um *array* de bytes no qual, antes dos bytes de cada elemento, é inserido (também em bytes) o tamanho desse mesmo elemento, de modo a que, quando este pacote for desserializado, o recetor saiba até que ponto deve ler de modo a não estender ou cortar partes referentes a diferentes elementos do pacote.

A lista de comandos que faz uso deste protocolo e deste tipo de pacotes são os seguintes:

- *register* - Utilizado exclusivamente na inicialização da conexão entre o FS_Node e o FS_Tracker, é utilizado para registar a existência do nodo e as informações referentes ao mesmo
- *update* - Utilizado para atualizar a lista de ficheiros de um nodo (*)
- *get* - Utilizado para fazer um pedido ao FS_Tracker de modo a que este responda com a lista de nodos que contém um determinado ficheiro (especificado pelo utilizador) e os IDs dos blocos que cada nodo contém
- *exit* - Utilizado para terminar a conexão entre um nodo e o FS_Tracker, removendo todas as informações relativas a esse mesmo nodo

(*) Dada a forma como decidimos implementar os diferentes aspetos deste projeto, o nosso programa utiliza o comando *update* de duas formas distintas:

- *update* manual - Quando um utilizador utiliza diretamente o comando *update*, existe a suposição de que foi adicionado manualmente um novo ficheiro ao diretório associado ao nodo, pelo que tanto o próprio nodo como o FS_Tracker devem ser informados e atualizados
- *update* automático - Ao longo do funcionamento do programa, certamente haverá troca de ficheiros entre vários nodos. Desta forma, o servidor vai sendo atualizado automaticamente a cada 0.5 segundos de forma a ter informação atualizada a todo e qualquer momento

3.2 FS Transfer Protocol

Em contraste com o que foi pedido e feito para o FS Track Protocol, este protocolo foi implementado sobre UDP, pelo que as estratégias utilizadas foram também elas diferentes, em certos aspetos. Ao contrário do que é feito em conexões TCP, o protocolo UDP não estabelece uma conexão dedicada entre quem envia e recebe mensagens. Posto isto, existe uma necessidade de garantir a fiabilidade de entrega das mensagens através de ACKs e *Checksums*, por exemplo.

Posto isto, as mensagens trocadas entre nodos através do FS Transfer Protocol serão do tipo Transfer_Packet, sendo este constituído pelos seguintes componentes:

- *String file_name* - Nome do ficheiro a ser tratado
- *int block_id* - ID do bloco a ser tratado
- *byte[] data* - Informação do bloco referido em bytes
- *int total_blocks* - Número total de blocos referentes ao ficheiro em questão
- *long checksum* - Número a ser utilizado posteriormente pelo recetor para verificar a integridade dos ficheiros

Mais uma vez, tal como para os Track_Packets, estas informações são serializadas para um *array* de bytes e só depois enviadas. A conversão é feita com os mesmos princípios descritos para o protocolo anterior.

Este protocolo será utilizado apenas através do comando *transfer*, apesar de, durante a execução deste comando, serem utilizados comandos do protocolo anterior (como *get* e *update*) de modo a obter e atualizar informações de forma suave ao longo do programa.

4 Implementação

4.1 Argumentos

4.1.1 FS Tracker

O FS Tracker não recebe argumento nenhum como input, pois não é necessário. Isto deve-se ao uso do DNS para descobrir o seu próprio nome e ao facto do Tracker estar sempre a ouvir na mesma *port* (9090).

4.1.2 FS Node

O FS Node recebe como argumentos a diretoria que irá usar para a partilha de ficheiros, o nome/IP do Tracker a que se pretende conectar e a *port* que devia utilizar para comunicar com o mesmo.

4.2 FS_Tracker

A implementação do FS_Tracker foi talvez o elemento mais simples do projeto, apesar de nos termos deparado com alguns problemas. Inicialmente, toda a informação relativa aos blocos era guardada no Tracker em forma de bytes, o que não era, de todo, uma boa solução. Ao longo do projeto fomos alterando este aspeto até chegarmos ao ponto atual, no qual as informações dos nodos são guardadas num *Map<String, Map<String, List<Integer»>*, no qual a chave do *Map* inicial é o endereço do nodo, a chave do segundo *Map* é o nome de cada ficheiro a ele associado, e a *List* contém os *IDs* dos blocos desses mesmos ficheiros. Apesar de ainda assim ser mais complexo do que gostaríamos, consideramos que é fácil entender como funciona.

O FS_Tracker, assim que iniciado, está sempre a correr e à escuta por conexões, sendo criada uma nova *thread* sempre que recebe um pedido de conexão por parte de um nodo. Assim que aceita esta conexão, fica permanentemente à espera que o nodo envie novos comandos a ser executados (até que eventualmente a conexão seja terminada).

Todos estes comandos (já referidos anteriormente) são bastante simples do lado do Tracker, visto que envolvem apenas alterar ou retirar informações do *Map* associado ao nodo que lhe enviou dito comando.

4.3 FS_Node

O FS_Node foi de longe o componente mais complicado de implementar, devido principalmente à quantidade de aspetos a ter em conta, visto que virtualmente todo o projeto envolve, de uma forma ou outra, o FS_Node. É relevante referir que cada nodo faz uso de um *Map<String, List<Integer»* de modo a guardar as informações relativas aos seus ficheiros.

4.3.1 FS_Node e FS_Tracker

No que diz respeito à conexão e interações entre estes dois componentes, tal como foi dito anteriormente, é utilizado o FS Track Protocol que funciona sobre TCP. Como também foi referido anteriormente, a primeira coisa a ser feita após a inicialização de um nodo é o estabelecimento da conexão entre este e o Tracker, sendo também enviado o comando *register*. A partir daqui todas as interações são operações simples que envolvem o envio e receção de informações já contidas no próprio nodo ou no Tracker. É de notar que, para o *get* e *exit*, no qual certas informações presentes no pacote não são necessárias (*Map<String, List<Integer» files*), é utilizado um "*placeholder*", de modo a não enviar informação desnecessária e excessiva.

4.3.2 FS_Node e Transferências de Ficheiros

Apesar de haver apenas um comando dedicado a esta componente do trabalho, verificamos também que a quantidade de pequenos aspetos a ter em conta ao longo do desenvolvimento da mesma veio a torná-la bastante complexa.

Assim que o nodo é criado e a conexão com o Tracker é estabelecida, é também criada uma nova *thread* no próprio nodo responsável por receber pacotes via FS Transfer Protocol. Por sua vez, sempre que um pacote é recebido, é criada uma nova *thread* para lidar com esse mesmo pacote. Para isto, utilizamos um *ExecutorService* e *newCachedThreadPool* de modo a criar e reutilizar *threads* automaticamente, conforme o necessário.

Inicialmente, é pedido ao utilizador para especificar o ficheiro a transferir. Após obter esta informação, utilizamos o comando *get* para obter, através do Tracker, os nodos no qual existem blocos deste mesmo ficheiro. A partir daqui, é criada uma nova *thread* para cada *ID*, responsável por enviar o pedido ao melhor nodo possível.

A forma como selecionamos o melhor nodo da rede para transferência de um dado bloco é bastante simples. Primeiro ordenamos a lista de nodos que contém o dado bloco por ordem crescente de número total de blocos, visto que existe uma menor probabilidade de receberem mais pedidos se não tiverem mais blocos. Seguidamente, para cada endereço, é feito um *Round Trip Time* (RTT), sendo o tempo que esta operação demora (em milissegundos) adicionado ao valor anterior. Através desta soma final, utilizamos o menor valor e fazemos o pedido de envio do bloco ao respetivo endereço. Esta solução não é a mais eficaz e é um ponto do projeto que gostaríamos de ter melhorado, apesar de não termos conseguido chegar a uma ideia de como melhorar este algoritmo.

De modo a distinguirmos se o pacote recebido é um pedido ou verdadeiramente um bloco que foi enviado, utilizamos o parâmetro *checksum* (já referido) como *flag*. Caso se verifique que o *checksum* tem o valor -1, o pacote será considerado um pedido, sendo o *file_name* e os restantes parâmetros referentes ao bloco a enviar (o parâmetro byte[] data estaria vazio de modo a não enviar informação desnecessária).

4.4 DNS

Para a implementação do serviço de resolução de nomes, decidimos utilizar o bind9 para criar a zona de DNS com os nodos e os servidores presentes no CORE, como foi proposto nas aulas práticas. Também decidimos implementar uma tabela para o *Reverse Lookup*, pois é utilizado no nosso trabalho para o nodo saber o seu próprio nome sem o receber como argumento. Esta implementação resultou de erros ao usar o *InetAddress.getLocalHost()*, o que nos levou a tentar arranjar uma solução alternativa. Esta é procurar por todas as *Network Interfaces* até encontrar o seu próprio IP e de seguida usar o *Reverse Lookup* para resolver o IP num nome.

4.5 Tratamento de erros de rede

Para lidar com a perda e duplicação de pacotes na comunicação entre nodos, decidimos implementar os seguintes sistemas

4.5.1 ACKs

Sempre que um nodo envia um pacote de transferência (através de UDP) para outro nodo, ele fica à espera de uma mensagem de reconhecimento (ACK) do nodo a que enviou a informação. Esta mensagem é simplesmente uma *String* com o valor "ACK". Enquanto o nodo que enviou o Transfer_Packet não conseguir ler o ACK de volta, tenta enviar outra vez. Isto acontece até finalmente receber o ACK ou até ultrapassar 5 tentativas de envio. Isto aplica-se tanto aos pacotes de pedido como aos com os dados do ficheiro, pois é necessário assegurar a receção de ambos os pacotes para garantir o bom funcionamento do sistema.

A nossa implementação faz exatamente isso, garantindo que o ficheiro é transferido na sua totalidade, mas tem alguns problemas em relação à quantidade de pacotes duplicados ou redundantes (o nodo que os recebe já os tem) enviados. Isto resulta em problemas quando tentamos enviar ficheiros maiores, pois o programa acaba por ocupar muita memória a enviar pacotes com blocos que o nodo já tem. Para mitigar este problema, seria necessário implementar uma forma de um nodo comunicar ao outro que não precisa de mais blocos, e este teria de ter algum método para suspender todos os Threads que estão a enviar pacotes para esse nodo.

4.5.2 Duplicação de Pacotes

A única forma que temos de lidar com pacotes duplicados (em específico os que têm blocos) é descartá-los caso o bloco recebido já esteja presente no ficheiro do nodo que o recebeu. Para otimizar o trabalho, seria necessário também tratar de pedidos duplicados, que resultam no envio de muitos dados redundantes.

4.5.3 Perda de Nodos

Caso um nodo, por alguma razão, fique incontactável a meio do processo de pedido de transferência de um bloco de um ficheiro, o nodo irá procurar outra vez qual é o nodo a que deveria fazer o pedido e continua em *loop* até conseguir enviar um pedido e receber um ACK de volta(como no método que determina o melhor nodo é verificado se os nodos são contactáveis, não é necessário remover o nodo incontactável do Map devolvido pelo *get*. Se não houver esse bloco na rede ou o nodo que está a fazer o pedido já o tiver, o *loop* é quebrado.

As limitações desta implementação são principalmente em assegurar que o ficheiro é completamente transferido, pois existem dois casos onde isso não se verifica: como foi dito antes, quando esse bloco não é encontrado na rede e quando o nodo a que foi feito o pedido sai do sistema depois de devolver o ACK ao nodo que fez o pedido. Para lidar com ambas as situações, podíamos ter implementado uma função que, após um *transfer*, verifica se há blocos em falta e faz um novo pedido ao servidor até um nodo que tenha esses blocos em falta se conecte, fazendo os pedidos de transferência ao mesmo. Com a implementação atual, é necessário fazer outra vez *transfer* do ficheiro para garantir que o ficheiro tem todos os blocos

4.6 Tratamento de Blocos

Inicialmente, o nosso programa guardava os blocos que recebia em memória, só os escrevendo para um ficheiro quando tivesse todos os blocos. Mesmo assim, os blocos tinham de se manter em memória para enviar para outros nodos. Como achamos esta implementação pouco eficiente, usamos o RAF (*RandomAccessFile*) para abrir o ficheiro num certo ponto. Assim, só precisamos de multiplicar o ID do bloco pelo tamanho dos blocos para saber em que byte do ficheiro a informação contida no pacote começa. Isto serve tanto para escrever dados recebidos como para ler dados para enviar. Assim, conseguimos reduzir a memória ocupada pelo programa substancialmente sem perder muita performance.

5 Testes e Resultados

Para testar o nosso trabalho, utilizamos a topologia do CORE fornecida pelos docentes, usando a máquina virtual também fornecida pelos mesmos. Na seguinte imagem conseguimos ver a topologia a correr:

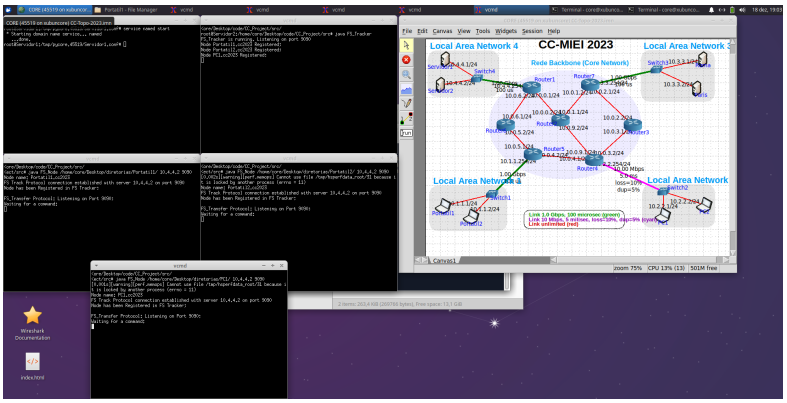

Figure 1: Topologia

Como podemos verificar na figura, decidimos utilizar o Servidor1 como o servidor de DNS e o Servidor2 como o FS_Tracker. Para fazer os testes, decidimos utilizar os nodos Portatil1, Portatil2 e PC1. Isto pois desta forma conseguimos testar diferentes casos de uso e verificar se o trabalho está a funcionar corretamente ou não.

Para verificar o comportamento do nosso trabalho diante de vários fatores, decidimos testar os seguintes casos:

- Transferência entre dois nodos sem perda/duplicação de pacotes (Portatil1 -> Portatil2)
- Transferência entre dois nodos com perda/duplicação de pacotes (Portatil1 -> PC1)
- Transferência de um nodo para dois em simultâneo (Portatil1 -> Portatil2 & PC1)
- Transferência de dois nodos para um em simultâneo (Portatil1 & Portatil2 -> PC1)
- Transferência e envio em simultâneo (Ocorre no PC1 e Portatil2)

De seguida, apresentamos os resultados destes testes, na mesma ordem que a anterior:

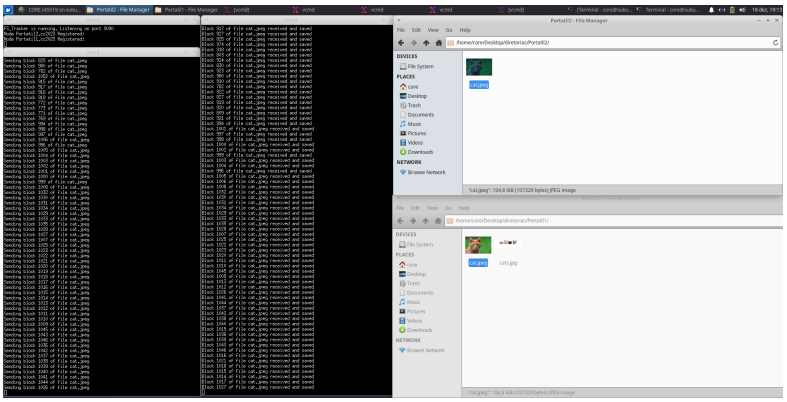

Figure 2: Transferência sem perdas/duplicação

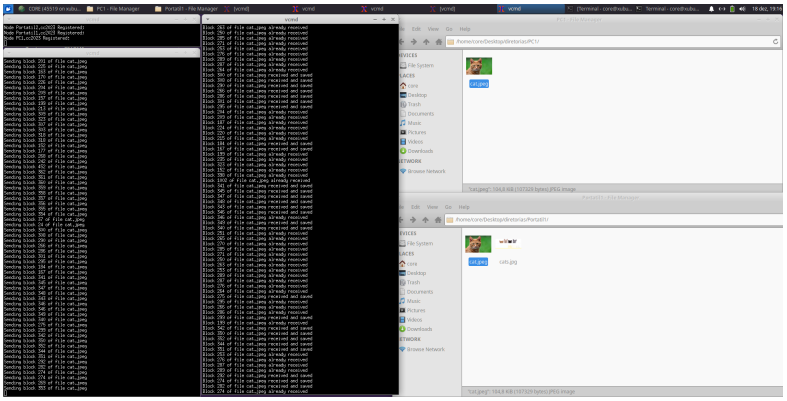

Figure 3: Transferência com perdas/duplicação

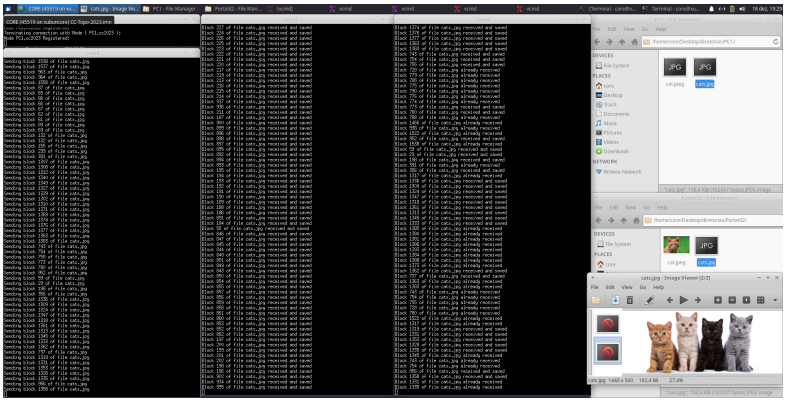

Figure 4: Transferência de um nodo por dois em simultâneo

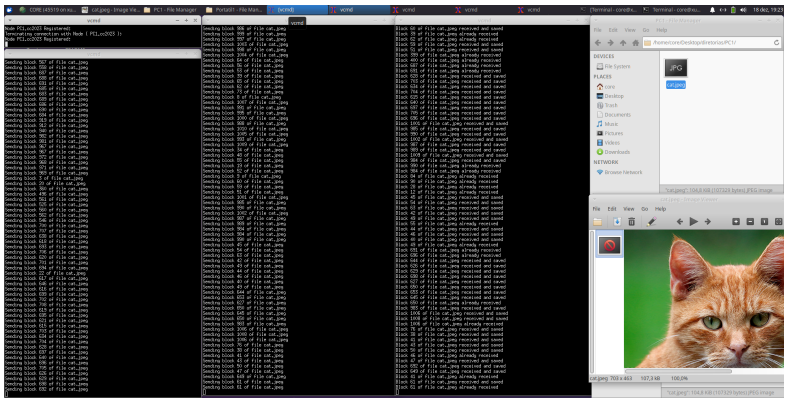

Figure 5: Transferência de dois nodos para um em simultâneo

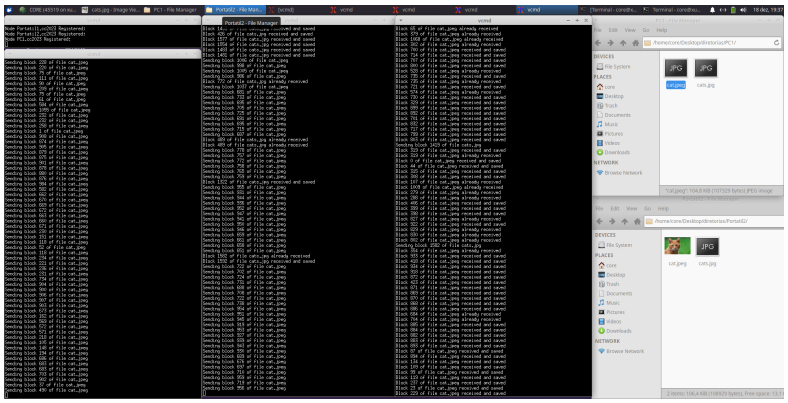

Figure 6: Envio e transferência em simultâneo

Conseguimos verificar que, em todos os casos, a partilha do ficheiro foi bem-sucedida, com a imagem a ter exatamente o mesmo tamanho e aspeto depois de enviada na sua totalidade. Também conseguimos verificar uma quantidade alta de pacotes repetidos/duplicados quando testamos com o nodo PC1 (o nodo que está na sub-rede com falhas), como tinha sido referido anteriormente no trabalho. Apesar disto, consideramos os testes um sucesso pois o programa assegura o nosso objetivo principal, partilhar o ficheiro na sua totalidade independentemente do caso.

6 Conclusão

No geral, todos os elementos do grupo sentem que o trabalho realizado foi não só interessante mas também extremamente útil para uma melhor compreensão dos conceitos nele envolvidos. Consideramos que, apesar de termos implementado todos os aspetos referidos no enunciado, existem componentes do trabalho que poderiam ter sido melhor explorados e desenvolvidos.

Posto isto, concluímos o aspeto com uma impressão positiva em relação ao trabalho realizado e esperamos ter a oportunidade de aprofundar e polir os conhecimentos obtidos no futuro.