Trabalho Prático N.º2 – Transferência rápida e fiável de múltiplos servidores em simultâneo

Diogo Barros[A100600], António Silva[A100533], Pedro Silva[A100745]

Universidade do Minho, Campus de Gualtar, Braga

Keywords: P2P, FS Track Protocol, FS Transfer Protocol, Nodos, Servidor

1 Introdução

Neste trabalho prático foi nos proposto a criação de um serviço de partilha de ficheiros peer-to-peer semelhante ao BitTorrent. Para isso o nosso programa tinha de conseguir reproduzir com sucesso dois protocolos: FS Track Protocol que guarda a informação de todos os nodos conectados e os seus ficheiros num servidor principal e o FS Transfer Protocol que permitiriam a partilha de ficheiros entre nodos com o uso do protocolo de transporte UDP.

Sendo assim, nós tivemos de tratar deste problema em três formas, um servidor que estaria maioritariamente encarregue em registar cada nodo e os ficheiros que estes têm, a forma como o nodo se regista no servidor e lhe informa dos seus ficheiros e por fim transformar o nodo também de certa forma num servidor para que este consiga partilhar os seus ficheiros entre outros nodos que estariam a funcionar da mesma forma. Além disso, tivemos de nos preocupar como procederiamos às divisões dos ficheiros para que a partilha seja executada. Para isso, quando um nodo se conectava ao servidor, a primeira função que ele executava depois do seu registo, seria a divisão dos seus files já em chunks.

Depois de termos tratado destes protocolos e da divisão dos ficheiros, faltava-nos ainda tratar da fragmentação dos ficheiros, pois não dava para mandar os chunks todos de uma vez. Para isso acrescentamos um limite de chunks por socket e caso faltem chunks por transferir, nós calculamo-los previamente e vamos fazendo continuamente a transferência até o ficheiro estar completamente transferido.

O nosso serviço funciona com uma UI básica. No servidor, mostra onde está ativo e em que porta e os nodos que se conectam e desconectam e cada nodo tem uma UI onde podemos pedir o download de ficheiros, que ficheiros este partilha, podemos apagar ficheiros dele e desconectá-lo do servidor.

Depois desta breve introdução do nosso trabalho, podemos proceder à demonstração da arquitetura da nossa solução com mais detalhe onde temos as várias funções implementadas e mais à frente faremos a especificação dos protocolos e como este foram implementados na linguagem que escolhemos: Java. Também demonstraremos um teste que fizemos de transferências entre 5 nodos e os seus resultados, tal como que trabalho futuro teria de ser feito neste projeto.

2 Arquitetura da Solução

Na criação da nossa solução, nós seguimos uma certa ordem de construção do programa:

- Começamos pela criação básica do nosso FS_Tracker e FS_Node onde estariam a criação de cada um, ou seja, o servidor e o nodo, tal como um Manager e um Node, onde ficam guardados os dados de cada um. Também criamos as Utils onde iremos guardar funções utilizadas ao longo de toda a solução.
- A seguir, procedemos à conexão dos nodos ao tracker e o tracker guardar a informação sobre cada nodo. Também implementamos uma pequena UI para o servidor e para cada nodo.
- Depois de já termos o servidor e os nodos, criamos o SeedingServer que transforma cada nodo também em si próprio num servidor. Com isso também passamos a adicionar os files a cada nodo, passando-os em argumento. Para pormos a solução mais limpa, tratamos de fechar o servidor da forma mais limpa possível.
- Após já termos colocado os files em cada nodo, passamos a poder pedir que ficheiro cada nodo está a partilhar com o uso de "Ficheiros Partilhados" na UI e começamos a formalizar a nossa forma de transferência de ficheiros primeiramente sem blocos, utilizando uma topologia criada por nós.
- Já com algum parte da transferência pronta, procedemos à criação dos chunks dos ficheiros, utilizando serializes e depois conseguimos colocar as transferências a funcionarem todas em paralelo.
- Para concluir o trabalho, fizemos alguns ajustes, sendo eles, adicionar documentação ao código, o servidor agora recebe updates sobre ficheiros novos em nodos e acrescentamos a nossa fragmentação aos ficheiros em blocos.

No final ficamos com estes ficheiros de código e os outros ficheiros são os que colocamos em cada um dos nossos nodos, tendo a maioria 94KB com texto em binário, excluindo o COD com 16 KB para testes mais rápidos.

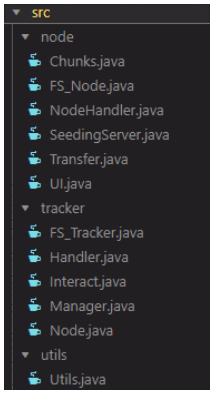

Fig. 1. Ficheiros de Código

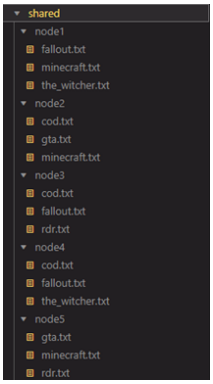

Fig. 2. Ficheiros partilhados

Esta foi a topologia que criamos para testar a nossa solução que tem um servidor e 5 nodos:

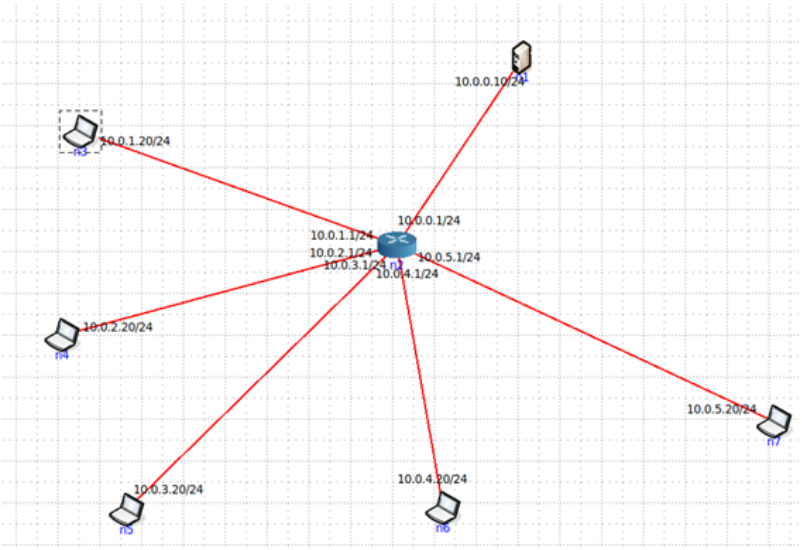

Fig. 3. Topologia utilizada

Conseguimos ver então que além da Pasta shared onde estão os ficheiros que colocamos em cada nodo, temos a Pasta src que tem todo o nosso código.

Esta pasta está dividida em 3 packages:

- **Node** - está mais encarregue nos nodos e na sua parte dos protocolos, principalmente em quase toda a parte das transferências de ficheiros
- **Tracker** - está encarregue na parte do servidor, guardar as informações de cada nodo e os pedidos de um cliente ao servidor, tal como a atualização de tudo que acontece
- **Utils** - trata de várias funções auxiliares de funcionamento geral do programa, por exemplo, todos os serializes e desserializes que usamos

No Package node temos 6 classes:

- **FS_Node** - Classe principal desta package, tem todas as funções principais de um nodo, desde o manuseamento dos ficheiros para blocos, do registo do nodo, do pedido de transferências, updates ao servers e adição e remoção de ficheiros
- **Chunks** - Classe que trata dos cálculos dos chunks para separação dos ficheiros
- **NodeHandler** - Classe que lida com os pedidos de um cliente e envia os ficheiros
- **SeedingServer** - Classe que trata de receber pedidos UDP de outros clientes
- **Transfer** - Classe que trata de toda a lógica por trás do pedido e recolha da transferência do ficheiro pretendido.
- **UI** - Classe que trata de toda a interface com o utilizador.

No Package tracker temos 5 classes:

- **FS_Tracker** - Classe principal desta package, tratando do servidor. Responsável por receber conexões e criar handlers para cada nodo conectado.
- **Handler** - Classe que recebe e lida com pedidos de um nodo conectado ao servidor, por exemplo, a sua conexão ao servidor
- **Interact** - Classe de interação com o servidor, observa separadamente por inputs para interação. Usada principalmente para fechar o servidor de maneira limpa.
- **Manager** - Classe que guarda todos os dados da rede. Inclui os dados dos nodos que estão conectados e algumas operações sobre eles.
- **Node** - Classe que guarda as informações de um nodo.

No Package utils somente temos a classe Utils que já foi explicada previamente.

3 Especificação dos protocolos propostos

3.1 FS Track Protocol

O nosso FS Track Protocol trata de todos os requisitos pedidos, mas de uma forma ligeiramente diferente:

- Conseguimos registar Nodos a qualquer momento, estes mantêm-se conectados ao Tracker e permitimos entrada e saída de nodos a qualquer momento.
- Sempre que um Nodo transfere um ficheiro de outro nodo ou apaga um ficheiro de si mesmo, o servidor mantêm-se atualizado acerca deste acontecimento.
- Acerca da localização de ficheiros, nós achamos que só seria necessária ser implementada nas transferências e sendo assim, sempre que um nodo quer transferir um ficheiro, o servidor comunica-lhe uma lista dos nodos onde este está.
- Como vou falar futuramente, na nossa técnica de transferência, nunca podem ficar a faltar blocos, quando se informa dos ficheiros, claramente informa-se dos blocos todos.

Acerca do formato das mensagens protocolares, nós utilizamos DataInputStream para fazer comunicações entre o Servidor e o Nodo.

Temos várias funções que participam neste protocolo, estando a grande maioria como já mencionei no Package tracker, por exemplo, o register() que está no Handler que trata do registo de um novo Nodo no Manager do Tracker que guarda todos os nodos existentes. Também temos nas Utils a Default Port usada no projeto inteiro: 9090.

Toda esta parte funciona com um ServerSocket que funciona sobre TCP, mantendo assim o FS Track Protocol a funcionar sobre TCP como é pedido.

Por fim sobre este protocolo, temos um pequena diagrama temporal de como funciona o registo de um novo nodo no servidor.

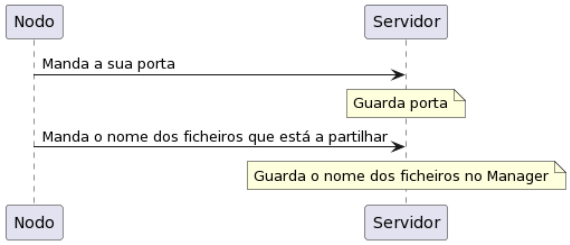

Fig. 4. Registo de um nodo no servidor

3.2 FS Transfer Protocol

O nosso FS Transfer Protocol também trata de todos os requisitos pedidos e nós implementamos a nossa forma de tratar de todos os blocos e como fazemos a fragmentação:

- O nosso programa aceita pedidos de blocos, em paralelo, de vários nodos, pois cada transferência é feita numa thread diferente.
- Quando um nodo quer fazer a transferência de um ficheiro, ele pede ao servidor em que nodos existe e, se houver mais do que um, ele faz a transferência em paralelo entre os diferentes nodos.
- Em termos de blocos perdidos, o nosso programa funciona de uma forma muito específica. Quando fazemos transferências de ficheiros, se passar do limite de blocos por datagramas e não houver nodos suficientes, simplesmente vai-se continuamente buscar o resto novamente aos mesmos nodos. E caso aconteça o raro caso de um ficheiro não ser completamente transferido, este simplesmente não comunica ao servidor que agora se encontra na pasta daquele nodo.

Em termos de formato dos nosso datagramas, os ficheiros são todos divididos em blocos, quando o nodo se conecta ao servidor. Depois quando é pedida a transferência desse ficheiro, primeiro verifica-se quantos nodos têm o ficheiro. Dependendo desse número, calcula-se quantos blocos vamos atribuir a cada nodo. Definimos também um limite de blocos que podem ser pedidos de uma só vez que depende do tamanho máximo dos packets UDP. Caso esse limite seja teoricamente excedido, limita-se o número de blocos para cada nodo ao máximo possível. Foi assim que tratamos da nossa fragmentação. Pode acontecer então o caso de haver chunks ainda não atribuídos, que são chunks que ficaram de fora devido a esse limite ou a divisão dos blocos não ser inteira. Isto significa que poderá ser necessário fazer mais do que um pedido ao mesmo nodo para a transferência do ficheiro ser completa.

Para ver o modo de funcionamento da transferência de um ficheiro, fizemos um pequeno diagrama temporal:

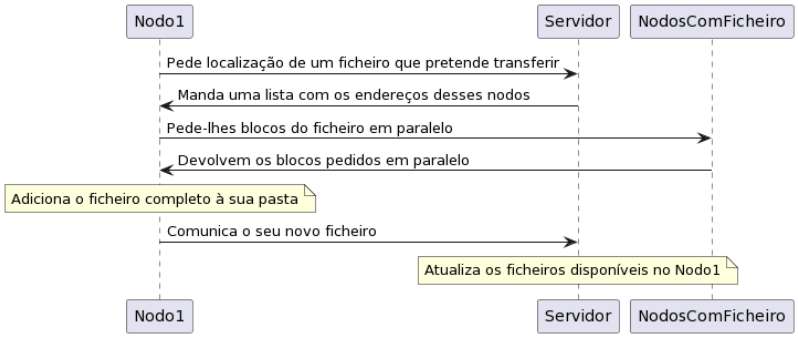

Fig. 5. Transferência de um ficheiro

4 Implementação

Nesta parte da implementação irei demonstrar algumas das funções principais do nosso programa. Uma das principais que vai ficar de fora são os serializes e desserializes, porque achamos que não é necessário demonstrar para o intuito desta cadeira, apesar de serem grande base da forma como transportamos dados.

Em primeiro lugar, temos as funções mains do FS_Node e do FS_Tracker, que é o código que é corrido quando se inicia o servidor e um novo nodo:

```
public static void main(String[] args) {
    filepath = args[0];
    try{
        register();
    }
    catch(IOException e){
        e.printStackTrace();
    }

    splitFiles();

    Thread server = new Thread(new SeedingServer(udpSocket));
    server.start();

    boolean end = false;
    while(!end){
        if(!Utils.checkConnection(dos)){
            disconnect();
            break;
        }
        int op = UI.menuPrincipal();
        if (op == 1) {
            String pedido = UI.menuDownload();
            requestDownload(pedido);
        }
        else if(op == 2){
            System.out.println("Ficheiros partilhados: " + String.join(delimiter: " ", sharedFiles.list()) + "\n");
        }
        else if(op == 3){
            String ficheiro = UI.menuApagar();
            apagaFile(ficheiro);
        }
        else if(op == 4){
            end = true;
            quit();
            System.out.println(x:"Desconectado com sucesso");
            return;
        }
    }
    System.out.println(x:"Conexão ao servidor perdida");
}
```

Fig. 6. Main do FS_Node

```
public class FS_Tracker {
    public static boolean state = true;
    public static ServerSocket serverSocket;
    Run | Debug
    public static void main(String[] args) throws IOException{
        serverSocket = new ServerSocket(Utils.DEFAULT_PORT);

        System.out.println("Servidor ativo em " + serverSocket.getInetAddress().getHostAddress() + " na porta " + serverSocket.getLocalPort());

        Manager manager = new Manager();

        Thread interactions = new Thread(new Interact(manager));
        interactions.start();

        while(state)
        {
            try{
                Socket clientSocket = serverSocket.accept();
                Thread handleRequest = new Thread(new Handler(clientSocket, manager));
                handleRequest.start();
            }
            catch(IOException e){
                System.out.println(x:"Servidor fechado");
            }
        }
    }
}
```

Fig. 7. Main do FS_Tracker

Na main do FS_Node, faz-se o registo do nodo, divide-se os ficheiros em chunks e abre-se a UI onde vamos selecionar o que acontecerá a seguir. Na main do FS_Tracker, iniciamos só o servidor e criamos o Manager e a thread das interactions para mantermos guardados os nodos e conseguirmos ver cada acontecimento.

Temos a função que trata do registo do nodo acabado de criar, tal como a função que faz o split dos files:

Fig. 8. Register do FS_Node

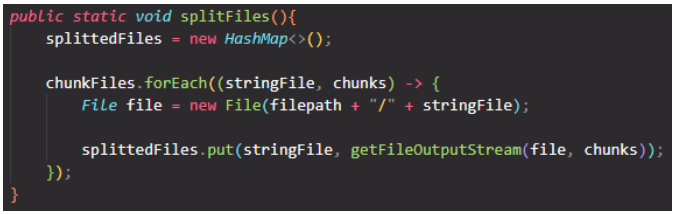

Fig. 9. SplitFiles do FS_Node

Temos a função que trata da UI de cada nodo:

Fig. 10. Função da UI do FS_Node

Temos as duas funções principais que inicializam a transferência, uma que seleciona os nodos e a ordem que inicia o pedido da transferência:

```
public static void selectNodes(String file, List<InetSocketAddress> nodosDisponveis, int chunks){
    int chunkAtual = 0;

    int chunkPorNodo = chunks / nodosDisponveis.size();
    if(chunkPorNodo > Utils.SOCKET_LIMIT){
        chunkPorNodo = Utils.SOCKET_LIMIT;
    }
    int missingChunks = chunks - (chunkPorNodo * nodosDisponveis.size());

    for(InetSocketAddress address : nodosDisponveis){
        Thread t = new Thread(new NewRunnable(file, address, chunkAtual, (chunkAtual + chunkPorNodo)));
        t.start();
    }
    int i = 0;
    while(missingChunks > 0){
        if(missingChunks >= chunkPorNodo){
            Thread t = new Thread(new NewRunnable(file, nodosDisponveis.get(i), chunkAtual, (chunkAtual + chunkPorNodo)));
            t.start();
        }
        else{
            Thread t = new Thread(new NewRunnable(file, nodosDisponveis.get(i), chunkAtual, (chunkAtual + missingChunks)));
            t.start();
        }
        i++;
        if(i == nodosDisponveis.size()){
            i = 0;
        }
        missingChunks -= chunkPorNodo;
    }
}
```

Fig. 11. selectNodes do Transfer

```
public static void requestDownload(String pedido){
    if (chunkFiles.containsKey(pedido)) {
        return;
    }
    List<InetSocketAddress> nodosDisponveis;
    try{
        dos.writeUTF("REQUEST " + pedido);
        dos.flush();

        int length = dis.readInt();
        byte[] data = new byte[length];
        dis.readFully(data);
        nodosDisponveis = Utils.deserializeList(data);

        if(nodosDisponveis.size() > 0) {
            int chunks = dis.readInt();
            Transfer.selectNodes(pedido, nodosDisponveis, chunks);

            chunkFiles.put(pedido, chunks);
            updateServer(chunkFiles);
        }
        else return;
        System.out.println(x:"Ficheiro transferido com sucesso\n");
    }
    catch(IOException | ClassNotFoundException e){
        e.printStackTrace();
    }
}
```

Fig. 12. RequestDownload do FS_Node

Temos o início da classe Manager e a run do Handler que tratam das partes principais do Tracker, por exemplo, guardar a informação de todos os nodos e tratar dos vários pedidos de um nodo.

```
public class Manager {
    public HashMap<InetSocketAddress, Node> nodes;

    /**
     * Função que inicializa a base de dados.
     */
    public Manager(){
        nodes = new HashMap<>();
    }

    /**
     * Função que regista um nodo novo na base de dados.
     * @param address Endereço do nodo a conectar.
     * @param request Pacote enviado pelo nodo com as informações necessárias para registo.
     * @param connection Objeto que lida com os pedidos do cliente a conectar.
     */
    public void registerNode(InetSocketAddress address, byte[] request, Handler connection){
        HashMap<String, Integer> files = new HashMap<>(Utils.deserializeMap(request));
        nodes.put(address, new Node(address, files, connection));
    }

    /**
     * Função que remove um nodo da base de dados
     * @param address Endereço do nodo a remover
     */
    public void removeNode(InetSocketAddress address){
        nodes.remove(address);
    }

    /**
     * Função que seleciona os nodos que possuem um ficheiro.
     * @param file Ficheiro pedido.
     * @return Lista com os endereços dos clientes.
     */
    public List<InetSocketAddress> getNodesFile(String file){
        List<InetSocketAddress> disponiveis = new ArrayList<>();
        this.nodes.forEach((address, node) -> {
            if(node.getFiles().containsKey(file)){
                disponiveis.add(address);
            }
        });
        return disponiveis;
    }
```

Fig. 13. Início da classe Manager

```
public void run(){
    try{
        register();
        System.out.println("Cliente " + clientAddress + " " + "conectado com sucesso");

        while(!end){
            String[] request = dis.readUTF().split(regex:" ");
            if(request[0].equals(anObject:"QUIT")){
                quit();
                manager.removeNode(clientAddress);
            }
            else if(request[0].equals(anObject:"REQUEST")){
                getNodes(request[1]);
            }
            else if(request[0].equals(anObject:"UPDATE")){
                int length = dis.readInt();
                byte[] initRequest = new byte[length];
                dis.readFully(initRequest);
                manager.updateNode(clientAddress, initRequest);
            }
        }
    }
    catch(IOException e){
        System.out.println(x:"Cliente desconectado inesperadamente");
        manager.removeNode(clientAddress);
        return;
    }
    System.out.println("Cliente " + clientAddress + " " + "desconectado com sucesso");
}
```

Fig. 14. Run do Handler do Tracker

Por fim temos as funções da classe Chunks que tratam do manuseamento inicial dos chunks dos ficheiros:

```java
public static HashMap<String, Integer> getChunks(String filepath, File sharedFiles){
    HashMap<String, Integer> chunks = new HashMap<>();

    String[] files = sharedFiles.list();
    for(String f: files){
        chunks.put(f, calculateChunks(filepath, f));
    }
    return chunks;
}

/**
 * Função que calcula a quantidade de chunks que um ficheiro vai ser separado
 * @param filepath Caminho para a pasta de partilha
 * @param f Nome do ficheiro a ser calculado
 * @return Número de chunks que o ficheiro f tem que ser dividido.
 */
public static int calculateChunks(String filepath, String f){
    return (int)(new File(filepath + "/" + f).length() / Utils.BLOCK_SIZE) + 1; // Um bloco extra para casos em que sobram bytes que nao formam um bloco completo.
}
```

Fig. 15. Funções da classe Chunks

Em termos de parâmetros, os principais estão guardados nas Utils (Default Port, Block Size, Margem Erro e Socket Limit):

```java
public class Utils {
    public static final int DEFAULT_PORT = 9090; // Porta genérica que todos os sockets se conectam se possível.
    public static final int BLOCK_SIZE = 5000; // Tamanho em bytes de um bloco.
    public static final int MARGEM_ERRO = 500; // Margem de erro de espaço para bytes das próprias estruturas.
    public static final int SOCKET_LIMIT = 65535 / BLOCK_SIZE;
```

Fig. 16. Utils do projeto

O nosso uso de bibliotecas foi somente focado nas já existentes no Java, sendo estas mais focadas nos conteúdos necessários, ou seja, DatagramSocket, DatagramPacket, InetSocketAddres, InetAddress, Socket, ServerSocket e vários Input/OutputStreams.

5 Testes e Resultados

Para terminar temos um teste específico que fizemos para demonstrar todas as funcionalidades a executar ao mesmo tempo.

Nesta primeira imagem, conseguimos ver o servidor ligado, todos os nodos conectados e os ficheiros que cada nodo partilha:

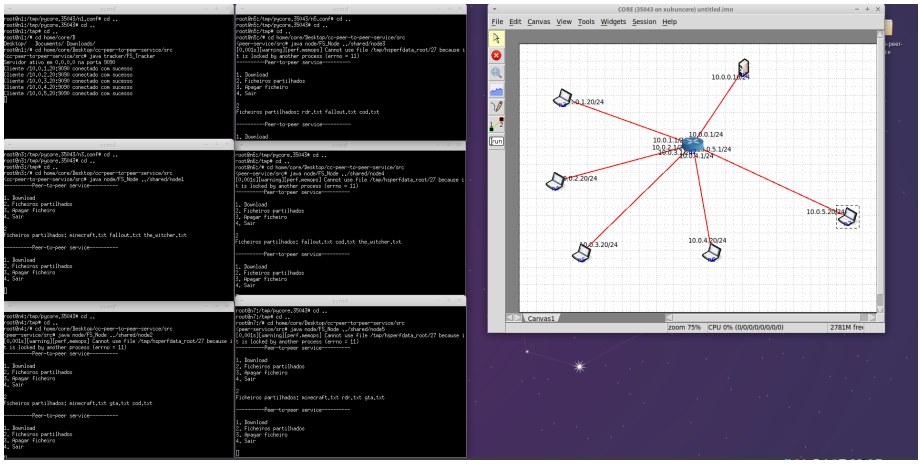

Fig. 17. Inicialização do teste e ficheiros partilhados por cada nodo

Dá para ver que os nodos 2, 3 e 4 têm o ficheiro "cod.txt". O próximo passo foi fazer a transferência do "cod.txt" para o nodo 1. Conseguimos ver que os nodos 2,3 e 4 enviaram o ficheiro, sendo que o nodo2 mandou duas vezes, pois foi ele que tratou das chunks que sobraram. Conseguimos ver também no nodo1 que ele agora também tem o "cod.txt" nos seus ficheiros partilhados.

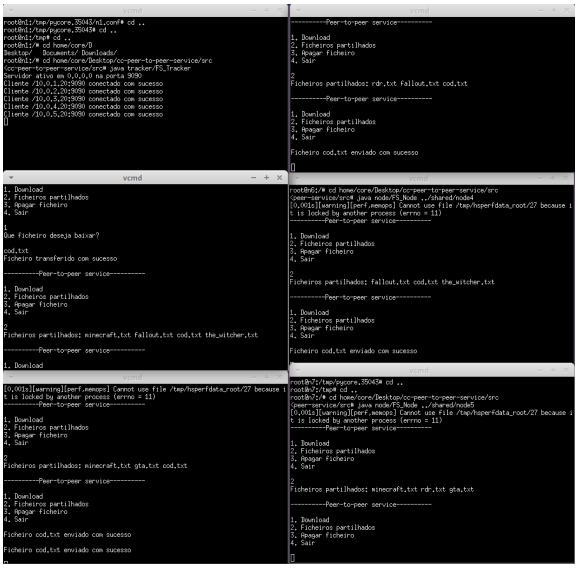

Fig. 18. Transferência do "cod.txt" para o Nodo1

Sendo assim, agora o único nodo que não tem o ficheiro é o nodo5. Para testar se os ficheiros partilhados do nodo1 foram atualizados no servidor, fizemos essa transferência. Conseguimos ver então que os 4 nodos enviaram blocos com sucesso, incluindo o nodo 1 e agora o nodo 2 só teve de enviar blocos uma vez, ou seja, neste caso, não houve blocos que sobraram, logo todos os nodos só tiveram de enviar uma vez.

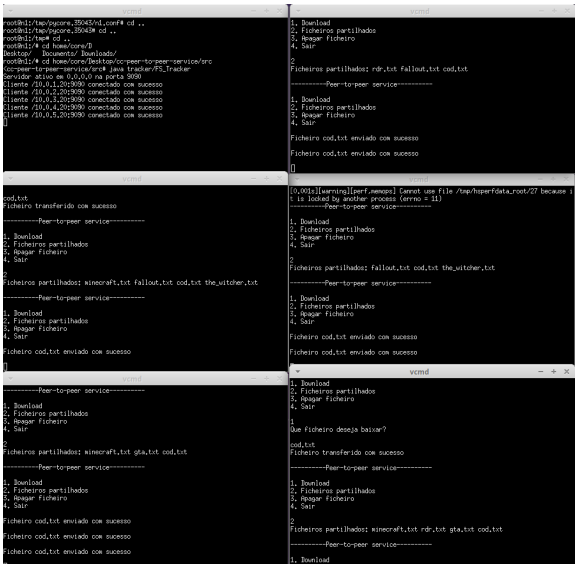

Fig. 19. Transferência do "cod.txt" para o Nodo5

Para finalizar o teste, fechamos cada nodo e o servidor de forma limpa, fechando assim todos os Sockets e todas as Threads existentes.

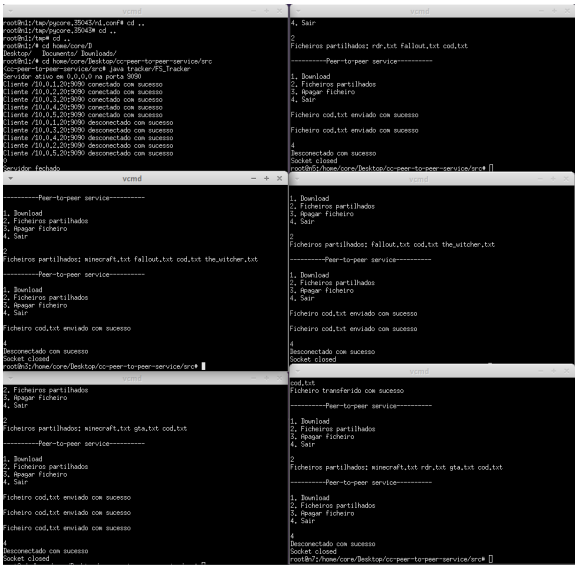

Fig. 20. Encerramento de todos os nodos e o servidor de forma limpa

6 Conclusões e Trabalhos Futuros

Depois de vermos o teste finalizado, além das principais implementações, especificações de cada protocolo que implementamos e a arquitetura da nossa solução em extensão, conseguimos concluir que temos um programa que funciona com boa qualidade.

Durante toda a produção deste programa, encontramos algumas dificuldades, sendo as principais: fechar todos os programas de forma limpa não deixando Sockets abertos, impossibilitando a abertura posterior de um novo servidor, não complicado, mas trabalhoso e a forma como organizavamos os vários blocos que eram recebidos em paralelo de modo a manter a forma original do ficheiro.

Em termos de trabalho futuro, nós não conseguimos implementar o Domain Name System(DNS) com o bind9, por isso seria o nosso próximo passo neste programa.

Em conclusão, o nosso programa consegue com sucesso implementar os dois protocolos principais requisitados e representar com sucesso como funciona uma rede peer-to-peer(P2P) na vida real. Sentimos que ficou um programa muito bem realizado e a única infelicidade foi não conseguirmos implementar o DNS.