Emulação de uma topologia real para teste de algoritmos distribuídos

Eduardo Rodrigues, Diogo Fonseca, Rui Teixeira

***Resumo*—Simulações são a base de grande parte do trabalho desenvolvido em sistemas distribuídos. No entanto, nem sempre essas simulações conseguem emular cenários da vida real. Como tal, apresentam resultados que podem ser incorretos quando submetidos a situações reais. Este projeto procurou desenvolver uma API de forma a introduzir uma *layer* entre as camadas de aplicação e de transporte capaz de emular as falhas e perdas inerentes a uma rede real. Dois métodos diferentes foram desenvolvidos, com base em trocas de mensagens e em técnicas de *routing*. Posteriormente, esses foram avaliados e comparados quanto à sua escalabilidade, tempo de execução e performance. Os resultados demonstraram que foi possível escalar a solução até um máximo de 1000 nós por máquina, sem introduzir demasiado atraso no tempo de convergência da mesma.**

I. Introdução

Um algoritmo distribuído pode apresentar um bom funcionamento na presença de uma rede ideal (i.e. sem falhas). No entanto, a performance baixa quando é submetido a falhas ou perdas. A maior parte destes algoritmos são simulados numa só máquina ou numa rede fechada, onde as falhas são praticamente inexistentes. Isto traduz-se num cenário diferente daquele a que os nós estarão submetidos se forem implementados numa rede variável, como a Internet.

De forma a melhorar o cenário de simulações, foram criados dois algoritmos que, com base na topologia da rede, provocam atrasos e falhas na comunicação entre nós. Estes programas foram escritos em JAVA e usam, na camada de transporte, troca de mensagens via UDP, fazendo uso da *class DatagramSocket*. Para simplificar o desenvolvimento das aplicações, foi criada uma API, denominada de *NodeDatagramSocket*, cuja utilização é semelhante à classe original. Ambas as metodologias desenvolvidas são compostas por um controlador e por vários nós que constituem a rede. O primeiro algoritmo desenvolvido baseou-se apenas em trocas de mensagens entre os nós. O controlador é responsável pela execução do *routing* e pela difusão da informação por todos os nós da rede de forma a que todos estejam cientes da topologia da mesma. O segundo, utiliza um ficheiro partilhado entre todos os intervenientes na rede, onde o controlador descreve a topologia. Neste caso, cada nó é responsável por executar o seu próprio *routing*. Estas duas metodologias foram avaliadas quanto à sua escalabilidade a nível de memória e CPU e quanto ao tempo necessário para convergência face a uma alteração na topologia.

II. Considerações

A. Caminho mais curto

Os algoritmos desenvolvidos simulam troca de mensagens entre nós com base na topologia, isto é, apesar de na realidade a troca ser feita ponto a ponto, o algoritmo simula todos os caminhos que a mensagem teria de percorrer até chegar ao destino pretendido. Como tal, foi necessário usar um algoritmo para descobrir o caminho mais curto entre os nós do grafo. A biblioteca algs4[1], fornecida pela Universidade de Princeton, foi utilizada como base do projeto para gerar os grafos e calcular o *routing*. No entanto, algumas alterações foram efetuadas de forma a alcançar os objetivos pretendidos. Nesta implementação, o peso atribuído às arestas representa o atraso da ligação entre dois vértices. Com base nesses pesos, o *routing* de cada vértice foi calculado utilizando o algoritmo de *Dijkstra*. Este foi escolhido por ser um dos mais famosos em grafos não direcionados e por ser a base de outros algoritmos de *routing* mais complexos, como o *OSPF* ou o *IS-IS*.

B. Falha nos nós

Numa rede real, um nó pode ficar demasiado lento, deixar de responder ou desligar-se da rede. Sendo estes acontecimentos importantes numa simulação, implementou-se um método capaz de ”matar”os nós a partir do controlador. Desta forma, o controlador

[1]http://algs4.cs.princeton.edu. [Acedido: 30-Dez-2017]

sabe sempre quando mudar a topologia sem ter de sobrecarregar a rede com *probes* constantes aos nós.

C. Vários nós por máquina virtual

A linguagem de programação JAVA tem a vantagem de ser agnóstica ao sistema operativo usado. Porém, os preços a pagar pelo uso de um *middleware* como o JVM são a velocidade e consumo de recursos. Logo, usar uma máquina virtual por nó limita o sistema, (secção IV). Para contornar esta limitação, a API oferece a possibilidade de se abrir vários nós numa JVM.

III. Implementações

Os algoritmos elaborados têm o mesmo objetivo. No entanto, o seu desenvolvimento focou-se em duas metodologias distintas. O primeiro basea-se em difusão de informação através de um ficheiro partilhado e de pequenas mensagens em *multicast*. O *routing* é feito localmente em cada nó. O segundo é totalmente baseado em troca de mensagens, onde o controlador executa o *routing* e difunde a informação via *unicast* para cada nó.

Em ambas as implementações, o controlador da rede tem como função construir o grafo que representa a topologia da rede e informar os nós de mudanças topológicas.

A. Ficheiro Partilhado

Nesta implementação, pretendeu-se um *routing* feito de forma distribuída, isto é, cada nó calcula o caminho mais curto para todos os outros nós da rede. Para tal, cada um necessita de ter conhecimento da topologia completa da rede. A figura 2 mostra um diagrama temporal do funcionamento deste algoritmo. Este inicia com o controlador a juntar-se a um grupo *multicast* e a criar o ficheiro partilhado com apenas a porta pela qual vai receber mensagens. Quando um nó inicia, junta-se ao grupo *multicast* e lê o ficheiro partilhado para ter informações sobre o controlador e a rede até ao momento. De seguida, envia uma mensagem de "HELLO" para o controlador. Esta mensagem é enviada através de uma porta especificada pelo utilizador, ou, se este não a escolher, obtida através da API do *DatagramSocket*. A porta será utilizada tanto para enviar como para receber mensagens entre nós ao nível da aplicação. O controlador guarda o endereço e a porta do nó que acabou de se juntar e adiciona um novo vértice, com uma ou mais arestas, ao grafo. Posteriormente, atualiza

```
TotalNumberOfNodes MASTERPORT MASTERIP
V_Id V_Id weigth IP port IP port
V_Id V_Id weigth IP port IP port
V_Id V_Id weigth IP port IP port
```

Figura 1. Estrutura do ficheiro partilhado. A primeira linha contêm o número total de nós da rede e o IP e porta pela qual o controlador está a receber as mensagens. Cada uma das seguintes linhas representa uma *edge* do grafo e é composta por dois nós, com os respetivos IPs e portas, e o peso da ligação

o ficheiro partilhado com todas as informações sobre a rede nesse instante. A estrutura do ficheiro pode ser observada na figura 1. De seguida, envia por *multicast* as novas *edges* que adicionou ao grafo, para que todos os nós fiquem a par das mudanças na topologia. Por último, cada nó calcula a sua tabela de *routing* através do algoritmo de *Dijkstra*. O processo repete-se para todos os novos nós que se queiram juntar à rede.

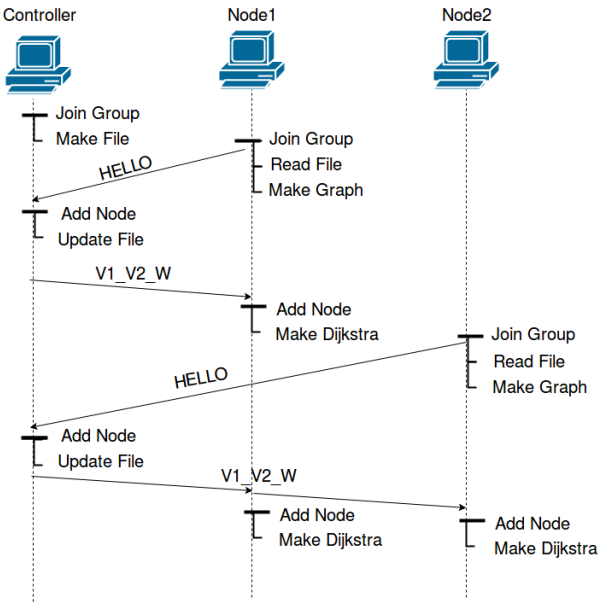

Figura 2. Diagrama temporal do funcionamento do algoritmo com base em ficheiro partilhado.

B. Troca de mensagens

Nesta segunda proposta, o *routing* é centralizado no controlador, ou seja, sempre que um nó se junta, o controlador calcula o caminho mais curto de cada nó para todos os outros nós presentes na rede. Desta forma, apenas o controlador tem total conhecimento da topologia rede. A figura 3 demonstra um diagrama temporal

da execução deste algoritmo. Primeiro, o controlador liga-se numa porta específica e espera a receção de mensagens dos nós. Quando um nó inicia fica à escuta em duas portas distintas, uma para comunicação com o controlador (agnóstica ao utilizador) e outra para comunicação com os restantes nós da rede, que pode, ou não, ser definida pelo utilizador. É necessário fornecer a cada nó o endereço e a porta do controlador, de forma a que lhe possa enviar uma mensagem. Esta mensagem contém o número da porta pela qual o nó espera receber comunicações com outros nós, ao nível da aplicação. Quando o primeiro nó inicia, envia a mensagem inicial e o controlador cria um grafo, ficando à espera de mais nós. Aparecendo o segundo nó, o controlador guarda os seus dados e adiciona um novo vértice ao grafo através de uma ou mais *edges*. Posteriormente, calcula-se a tabela de caminhos mais curtos para ambos os nós, utilizando o algoritmo *Dijkstra*. Por fim, é enviada para cada nó, via *unicast*, uma tabela com o atraso e a taxa de erros associados à ligação com os restantes nós. Este processo repete-se sempre que é inserido um novo nó. A tabela enviada a cada nó é realizada com recurso a um mapa (chave, valor), em que a chave é um objeto do tipo *NodeType* e o valor é um vetor com o atraso e a taxa de erros. O *NodeType* contém um ID fornecido pelo controlador, o endereço do nó, a porta de comunicação entre nós (ao nível da aplicação) e a porta para receber dados do controlador. Sempre que um nó recebe esta tabela, limpa a sua tabela e copia os valores da nova.

IV. Resultados

Os algoritmos descritos na secção III têm como objetivo ser a base de outros algoritmos distribuídos, como tal é importante estudar qual o *overhead* que podem introduzir nesses algoritmos. Para isso, foi estudado qual o impacto do aumento do número de nós na memória, no CPU e no tempo de convergência face a uma mudança na topologia na rede. A primeira experiência procurou apenas avaliar a escalabilidade dos algoritmos sem qualquer aplicação a executar "em cima" da biblioteca. Posteriormente foi avaliada a performance na presença de uma aplicação. As experiência foram repetidas com 1, 5 e 10 nós por JVM. De forma a que os resultados possam ser comparados, todas as simulações foram executadas no mesmo computador com 8 GB de RAM e de SWAP, Intel i5 6300HQ e Ubuntu.

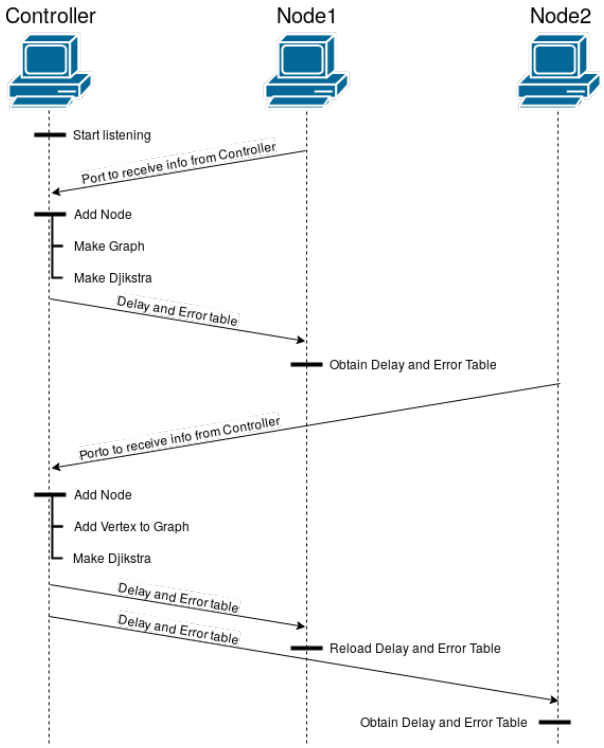

Figura 3. Diagrama temporal do funcionamento do algoritmo com base em troca de mensagens.

A. Memória

As figuras 4 e 5 mostram a evolução da memória com o aumento do número de nós na rede para os dois algoritmos. Pode observar-se que, em ambos os casos, foi atingido o limite e quanto maior o número de nós por JVM, menor é o declive, logo maior é a escalabilidade. Com um nó por JVM apenas se conseguiu obter cerca de 250 nós ativos no sistema. No entanto, com 5 ou 10 nós por JVM, o número máximo de nós aumentou para cerca de 1000 no algoritmo do ficheiro partilhado e 1200 no de trocas de mensagens. No caso do algoritmo da troca de mensagens (secção III-B) o sistema falhou porque a mensagem com o *routing* dos nós tornou-se demasiado grande para ser transmitida via UDP.

B. CPU

As figuras 7 e 6 mostram a evolução da carga no CPU com o aumento do número de nós. Em ambos os casos os valores são baixos, sendo que no algoritmo da

troca de mensagens se observa que quantos mais nós por JVM menor é a carga no CPU.

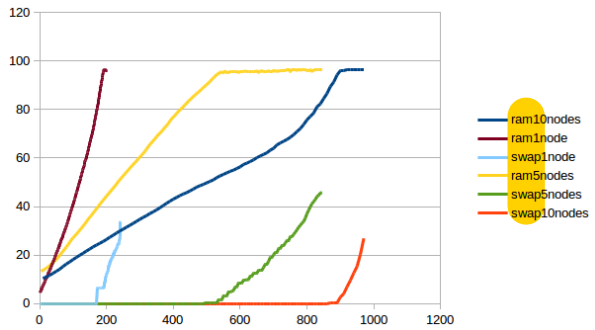

Figura 4. Partilha de ficheiro - Percentagem de memória usada.

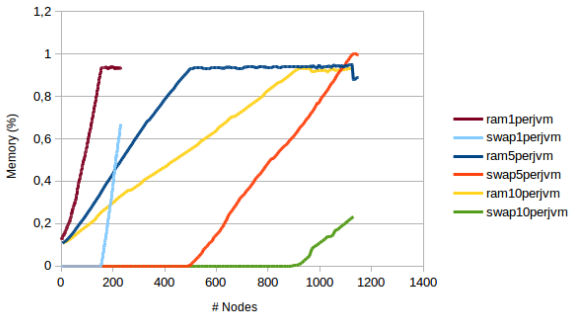

Figura 5. Troca de mensagens - Percentagem de memória usada.

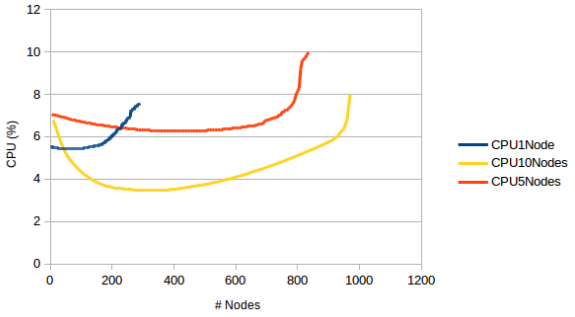

Figura 6. Partilha de ficheiro - Percentagem de CPU usado.

C. Tempo de convergência

As figuras 8 e 9 mostram o impacto do número de nós na rede no tempo necessário para convergir após uma alteração na topologia.

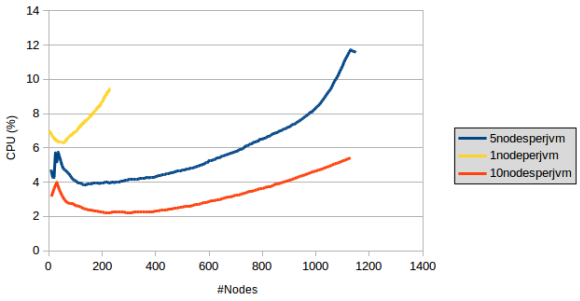

Figura 7. Troca de mensagens - Percentagem de CPU usado.

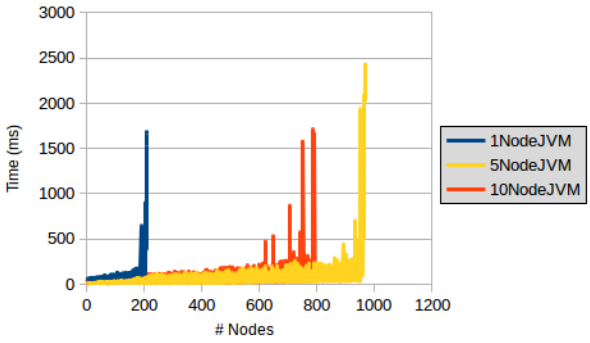

Figura 8. Ficheiro partilhado - Tempo de convergência após uma alteração na topologia.

No segundo gráfico é apenas apresentado o caso de 5 nós por JVM pois o gráfico resultante da reunião das três experiências era pouco percetível. Este foi escolhido por ser o que apresentava melhores resultados. Os outros gráficos podem ser observados no anexo A. No entanto, mesmo sendo o melhor caso, tem uma duração muito superior ao algoritmo de ficheiro partilhado. Por exemplo, com 600 nós, o tempo de convergência para 5 nós por JVM é de cerca de 2 segundos, enquanto que no algoritmo de ficheiro partilhado é cerca de 250 milissegundos.

D. Ficheiros partilhados versus Troca de mensagens

As duas implementações tem vantagens e desvantagens, sendo que a de ficheiros partilhados é mais rápida e mais escalável, ou seja, boa para ser utilizada numa única máquina, ou numa rede com sistema de ficheiros partilhado. Quando se tem a necessidade de usar várias máquinas e a configuração do sistema de ficheiros é demasiado custosa, o método da troca de mensagens torna-se útil, pois não requer configurações.

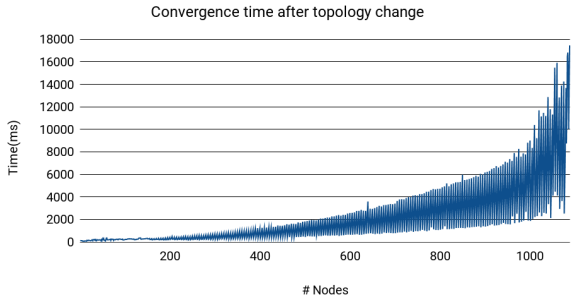

Figura 9. Troca de mensagens - Tempo de convergência com cinco nós por JVM.

Porém, como é feito à base de troca de mensagens UDP, que têm um tamanho máximo, só pode ter por volta de 1200 nós. Para além disso, a velocidade degrada-se consideravelmente com o aumento do número de nós, pois é o Controlador que executa o algoritmo de *routing*, ao contrário do algoritmo de ficheiros partilhados, em que este é feito de forma distribuída, aproveitando as possibilidades *multi-thread* do sistema.

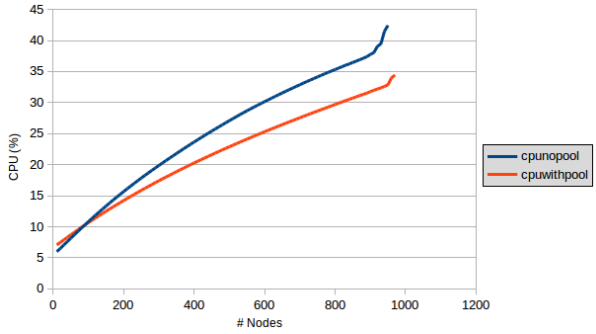

Figura 10. Thread pool vs full thread based - Percentagem de CPU usado com dez nós por JVM.

E. Teste com aplicação

Depois de avaliar os resultados das secções anteriores concluiu-se que o algoritmo do ficheiro partilhado tem menos limitações e, como tal, foi escolhido para proceder com as avaliações de *performance*. Nesta experiência, desenvolveu-se uma aplicação que utiliza o algoritmo para trocar mensagens entre nós da rede de forma periódica. Foram efetuadas duas abordagens, uma com a criação de um *thread* por mensagem enviada e outra com a utilização de uma *Thread pool*. Como é possível verificar na figura 10, a utilização do CPU

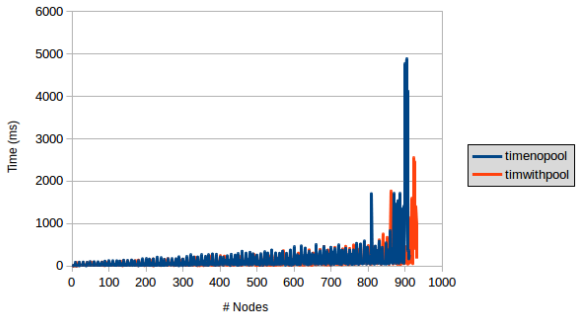

Figura 11. Thread pool vs full thread based - Tempo de convergência com dez nós por JVM.

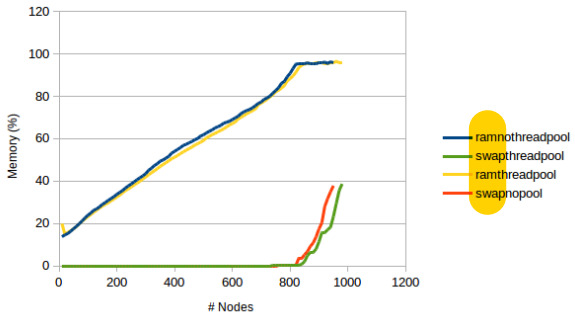

Figura 12. Thread pool vs full thread based - Percentagem de memória usada com dez nós por JVM.

aumentou bastante em comparação com os resultados anteriores. Também se pode concluir que o uso um sistema *Thread pool* é vantajoso. Porém, o tempo de convergência e o uso de memória, figuras 11 e 12, respetivamente, não aumentaram significativamente em relação aos resultados anteriores e são praticamente idênticos em ambas as implementações.

V. Conclusão

Este projeto foi desenvolvido com o propósito de tornar as simulações de algoritmos distribuídos mais próximas da realidade. Com a API desenvolvida é possível obter uma rede emulada que introduz falhas e perdas na comunicação com base na topologia. Os resultados demonstraram que a API pode ser utilizada sem introduzir demasiado *overhead* na aplicação.

Contribuições: Diogo (40%): Algoritmo de troca de mensagens, interfaces de cliente, simulações. Eduardo (40%): Algoritmo de partilha de ficheiro, preparação de topologia e Dijkstra, simulações. Rui (20%): Thread pool, envio de mensagens em *burst*, simulações.

ANEXO A
TROCA DE MENSAGENS - OUTROS GRÁFICOS

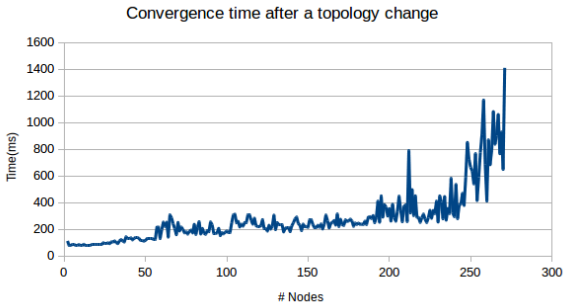

Figura 13. Troca de mensagens - Tempo de convergência com um nó por JVM.

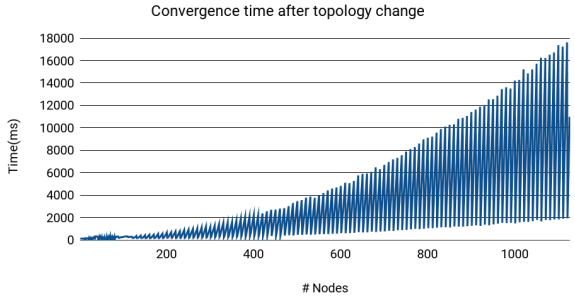

Figura 14. Troca de mensagens - Tempo de convergência com dez nós por JVM.

ANEXO B
API DO NODEDATAGRAMSOCKET

void receive(packet): espera um um pacote UDP.
void send(packet): envia um pacote UDP com atraso e erros.
DatagramSocket getSocket(): Retorna o socket do nó.
int getId(): Retorna o ID do nó.
int getPort(): Retorna a porta do nó.
int getIdByAddress(): Retorna o ID dum nó conhecido por endereço:porta.
int getPortById(int id): Retorna a porta dum nó conhecido pelo ID.
String getIpById(int id): Retorna o endereço dum nó conhecido pelo ID.
void printNodesMap(): Mostra os nós conhecidos e as suas caracteristicas.
String getMyAddress(): Obtém o endereço associado ao nó.
double getInitTime(): Obtém o momento, em milissegundos, em que o nó fez o pedido para pertencer à rede.

A. Partilha de ficheiros - construtor

DatagramSocket(int port): Usando uma porta específica.
DatagramSocket(): Usando uma porta aleatória.

B. Troca de mensagens - Construtor

DatagramSocket(int port, String masterHostname, int masterPort): Usando uma porta específica.
DatagramSocket(String masterHostname, int masterPort): Usando uma prota aleatória.

ANEXO C
INTERFACE COM O CLIENTE E APLICAÇÃO DE TESTES

```
[diogo@localhost SDIS-Projet]$ ./launch_controller.sh
Overlay Network Controller - Emulator of a real network
help
        help: Displays this message [help]
        close: Closes the overlay network or a specific node [close i]
        draw: Draw the graph in a separate window [draw]
        dijkstra: Show nodes connections with delays [dijkstra]
        network: Show network nodes list [network]
```

Figura 15. Demonstração da interface do controlador.

```
[diogo@localhost SDIS-Projet]$ ./launch_client.sh
192.168.1.80:49357 | delay port: 35983
192.168.1.80:49357 recebeu o ID: 2
(2) adquiriu lista de atrados em 86.0ms
Recebido de 1: ola
msg 1 ola
Enviado: ola
Falhou envio para 1
msg 1 ola
Enviado: ola
help
        help: Displays this message [help]
        msg: Send a message to node i [msg <address> <port> <text>]
        whoami: Return the IP:PORT of node [whoami]
        delay: Show delays for other nodes [delay]
```

Figura 16. Demonstração da aplicação de testes.